Herbert Schildt

C#

Traducere de Bogdan Pisai

Teora

Titlul original: **C#: A Beginner's Guide**

Teora
Calea Moșilor nr. 211, sector 2, București
fax: 01/210.38.28
e-mail: teora@teora.kappa.ro

Teora – Cartea prin poștă
CP 79-30, cod 72450 București, România
tel: 01/252.14.31
e-mail: cpp@teora.kappa.ro

Copertă: Gheorghe Popescu
Tehnoredactare: Techno Media
Director Editorial: Diana Rotaru

Președinte: Teodor Răducanu

NOT 5087 CAL C#
ISBN 973-20-0083-X

Printed in Romania

Cuprins

Despre autor

Herbert Schildt este primul autor în domeniul programării pe plan mondial. Este o autoritate în limbajele C, C++, Java şi C# şi un programator Windows experimentat. Lucrările sale în domeniul programării s-au vândut în peste 3 milioane de exemplare în toată lumea, fiind traduse în toate limbile de circulaţie internaţională. Este autorul a numeroase lucrări de succes, printre care „*C – Manual complet*", „*C++ – Manual complet*" (apărute la Editura Teora), „*Java 2 – Manual complet*", „*Java 2 – Manualul începătorului*", „*Programarea în Windows 2000 de la fundamente la noţiuni avansate*". Schildt a obţinut titlul de Master în Calculatoare la Universitatea Illinois.

Prefaţă

În ultimii ani, programarea a suferit o schimbare de paradigmă de la un peisaj dominat de sisteme izolate la un mediu on-line, în care sistemele sunt conectate în reţea. În epoca în care pentru foarte mulţi „reţeaua este calculatorul", limbajele de programare, sistemele de operare şi instrumentele de dezvoltare s-au străduit să ţină pasul. Limbajul C# a fost inventat ca răspuns la necesitatea unui limbaj de programare modern, care să corespundă cerinţelor lumii interconectate.

C# reprezintă pasul următor în evoluţia limbajelor de programare. Limbajul a preluat rezultatele cele mai bune din trecut, încorporând totodată şi ultimele descoperiri din proiectarea limbajelor de programare moderne. De exemplu, C# a împrumutat facilităţi atât din Java, cât şi din C++ – două din cele mai importante limbaje pe plan mondial. În acelaşi timp, C# a venit cu inovaţii proprii, cum ar fi delegările şi indexările. Datorită faptului că C# utilizează arhitectura .NET, codul rezultat este extrem de portabil şi permite programarea în limbaj mixt. De exemplu, componentele software create utilizând C# sunt compatibile cu codul creat în alte limbaje de programare, atât timp cât acesta este destinat tot arhitecturii .NET.

Scopul acestei lucrări este de a vă învăţa noţiunile fundamentale ale programării în C#. Lucrarea utilizează o abordare pas cu pas, completată cu numeroase exemple, teste propuse şi proiecte. Parcurgerea nu cere o experienţă anterioară în domeniul programării. Lucrarea începe cu chestiuni fundamentale, ca de exemplu compilarea şi rularea programelor C#. Sunt apoi discutate cuvintele cheie, facilităţile şi construcţiile care alcătuiesc limbajul C#. La sfârşitul lucrării, veţi stăpâni bine fundamentele programării în C#.

Este important să menţionăm de la început că această carte este doar un punct de pornire. Programarea în C# înseamnă mai mult decât cuvintele cheie şi sintaxa care defineşte limbajul. Aceasta implică utilizarea unui set sofisticat de biblioteci denumit „biblioteca de clase a arhitecturii .NET" (.NET Framework Class Library în original). Biblioteca de clase a arhitecturii .NET este foarte voluminoasă şi discutarea acesteia ar necesita o lucrare separată. Chiar dacă unele din clasele definite în această bibliotecă sunt prezentate în lucrarea de faţă, din cauza spaţiului restrâns cele mai multe nu sunt. Pentru a deveni un programator de elită în C#, trebuie să stăpîniţi foarte bine şi această bibliotecă. După parcurgerea acestei cărţi, veţi dobândi însă cunoştinţele necesare abordării oricărui alt aspect legat de C#.

Un ultim punct de vedere: C# este un limbaj nou. Ca şi toate celelalte limbaje de programare noi, C# va trece printr-o perioadă de dezvoltare şi modificare. Va fi necesar să urmăriţi apariţia unor noi facilităţi şi tehnici. De asemenea, nu trebuie să fiţi surprinşi dacă anumite aspecte se modifică în timp. Istoria limbajului C# este abia la început.

Organizarea lucrării

Această lucrare reprezintă un ghid struturat în secțiuni de lungimi sensibil egale, fiecare secțiune fiind bazată pe cele anterioare. Lucrarea conține 12 module, fiecare dintre acestea prezentând câte un aspect al limbajului C#. Aceasta este o lucrare unică, deoarece conține câteva elemente speciale care vă ajută să consolidați ceea ce ați învățat.

Scopuri

Fiecare modul începe cu un set de scopuri, care vă prezintă ceea ce veți învăța.

Verificarea cunoștințelor

Fiecare modul se încheie cu un test individual care vă permite să vă verificați cunoștințele însușite. Răspunsurile testelor se găsesc într-o anexă a lucrării.

Exerciții la minut

La sfârșitul fiecărei secțiuni majore sunt prezentate exerciții scurte care verifică înțelegerea noțiunilor esențiale din cadrul secțiunii. Răspunsurile acestor întrebări se găsesc în partea de jos a paginii.

Sfatul expertului

Casetele cu „sfatul expertului" se găsesc împrăștiate pe tot parcursul lucrării. Acestea conțin informații suplimentare sau comentarii interesante despre anumite subiecte. Modalitatea de prezentare este sub formă de întrebare-răspuns.

Proiecte

Fiecare modul conține unul sau mai multe proiecte care vă arată cum puteți aplica ceea ce învățați. Acestea sunt exemple reale, pe care le puteți utiliza ca puncte de pornire pentru propriile dumneavoastră programe.

Nu se cere experiență anterioară în programare

Această lucrare nu presupune că aveți o experiență anterioară în domeniul programării. În consecință, puteți utiliza cartea chiar dacă nu ați programat niciodată. Desigur că, după cum stau lucrurile în momentul de față, majoritatea cititorilor au cel

puțin un minimum de experiență în programare. Pentru majoritatea, această experiență anterioară este în C++ sau în Java. După cum veți învăța, C# este înrudit cu ambele limbaje. Prin urmare, dacă știți deja C++ sau Java, veți putea învăța C# cu ușurință.

Software necesar

Pentru compilarea și rularea programelor din această carte, aveți nevoie de Visual Studio.NET 7 (sau o versiune mai recentă) și arhitectura .NET trebuie să fie instalată pe calculatorul dumneavoastră. Programele din această carte au fost testate utilizând versiunea Visual Studio.NET 7.0, Beta 2.

Nu uitați: codul este disponibil pe Web

Codul sursă pentru toate exemplele și proiectele din această carte este disponibil gratuit pe Web la adresa **www.osborne.com**.

Lucrări recomandate pentru aprofundare

„*C#*" este poarta dumneavoastră de intrare către seria de lucrări de programare realizată de Herb Schildt. Iată alte câteva lucrări care vă pot fi utile.

Pentru a învăța mai mult despre C#, încercați:

- „*C# – Manual complet*"

Pentru a învăța limbajul C++, aceste lucrări vă vor fi în mod deosebit de ajutor:

- „*C++ – Manual complet*" (apărută la Editura Teora)
- „*C++ – Manualul începătorului*"
- „*Învățați singuri C++*"
- „*C++ de la fundamente la noțiuni avansate*"
- „*Programarea STL de la fundamente la noțiuni avansate*"
- „*Programarea C/C++ – arhive adnotate*"

Pentru învățarea programării în Java, vă recomandăm următoarele:

- „*Java 2 – Manualul începătorului*"
- „*Java 2 – Manual complet*"
- „*Java 2 – Manualul programatorului*"

Pentru a învăța programarea sub mediul Windows, propunem următoarele lucrări ale lui Herbert Schildt:

- „*Programarea Windows 98 de la fundamente la noțiuni avansate*"
- „*Programarea Windows 2000 de la fundamente la noțiuni avansate*"

Organizarea lucrării

Această lucrare reprezintă un ghid struturat în secțiuni de lungimi sensibil egale, fiecare secțiune fiind bazată pe cele anterioare. Lucrarea conține 12 module, fiecare dintre acestea prezentând câte un aspect al limbajului C#. Aceasta este o lucrare unică, deoarece conține câteva elemente speciale care vă ajută să consolidați ceea ce ați învățat.

Scopuri

Fiecare modul începe cu un set de scopuri, care vă prezintă ceea ce veți învăța.

Verificarea cunoștințelor

Fiecare modul se încheie cu un test individual care vă permite să vă verificați cunoștințele însușite. Răspunsurile testelor se găsesc într-o anexă a lucrării.

Exerciții la minut

La sfârșitul fiecărei secțiuni majore sunt prezentate exerciții scurte care verifică înțelegerea noțiunilor esențiale din cadrul secțiunii. Răspunsurile acestor întrebări se găsesc în partea de jos a paginii.

Sfatul expertului

Casetele cu „sfatul expertului" se găsesc împrăștiate pe tot parcursul lucrării. Acestea conțin informații suplimentare sau comentarii interesante despre anumite subiecte. Modalitatea de prezentare este sub formă de întrebare-răspuns.

Proiecte

Fiecare modul conține unul sau mai multe proiecte care vă arată cum puteți aplica ceea ce învățați. Acestea sunt exemple reale, pe care le puteți utiliza ca puncte de pornire pentru propriile dumneavoastră programe.

Nu se cere experiență anterioară în programare

Această lucrare nu presupune că aveți o experiență anterioară în domeniul programării. În consecință, puteți utiliza cartea chiar dacă nu ați programat niciodată. Desigur că, după cum stau lucrurile în momentul de față, majoritatea cititorilor au cel

puțin un minimum de experiență în programare. Pentru majoritatea, această experiență anterioară este în C++ sau în Java. După cum veți învăța, C# este înrudit cu ambele limbaje. Prin urmare, dacă știți deja C++ sau Java, veți putea învăța C# cu ușurință.

Software necesar

Pentru compilarea și rularea programelor din această carte, aveți nevoie de Visual Studio.NET 7 (sau o versiune mai recentă) și arhitectura .NET trebuie să fie instalată pe calculatorul dumneavoastră. Programele din această carte au fost testate utilizând versiunea Visual Studio.NET 7.0, Beta 2.

Nu uitați: codul este disponibil pe Web

Codul sursă pentru toate exemplele și proiectele din această carte este disponibil gratuit pe Web la adresa **www.osborne.com**.

Lucrări recomandate pentru aprofundare

„*C#*" este poarta dumneavoastră de intrare către seria de lucrări de programare realizată de Herb Schildt. Iată alte câteva lucrări care vă pot fi utile.

Pentru a învăța mai mult despre C#, încercați:

„*C# – Manual complet*"

Pentru a învăța limbajul C++, aceste lucrări vă vor fi în mod deosebit de ajutor:

„*C++ – Manual complet*" (apărută la Editura Teora)
„*C++ – Manualul începătorului*"
„*Învățați singuri C++*"
„*C++ de la fundamente la noțiuni avansate*"
„*Programarea STL de la fundamente la noțiuni avansate*"
„*Programarea C/C++ – arhive adnotate*"

Pentru învățarea programării în Java, vă recomandăm următoarele:

„*Java 2 – Manualul începătorului*"
„*Java 2 – Manual complet*"
„*Java 2 – Manualul programatorului*"

Pentru a învăța programarea sub mediul Windows, propunem următoarele lucrări ale lui Herbert Schildt:

„*Programarea Windows 98 de la fundamente la noțiuni avansate*"
„*Programarea Windows 2000 de la fundamente la noțiuni avansate*"

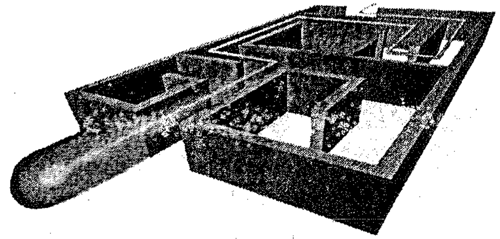
„*Programarea MFC de la fundamente la noțiuni avansate*"
„*Programarea Windows – arhive adnotate*"

Dacă doriți să vă însușiți limbajul C, care constituie baza întregii programări moderne, următoarele lucrări vă vor fi de ajutor:

„*C – Manual complet*" (apărută la Editura Teora)
„*Învățați singur C*"

Modulul 1

Fundamentele limbajului C#



Scopurile acestui modul

- **Înțelegerea evoluției limbajului C#**
- **Cunoașterea modului în care C# utilizează arhitectura .NET**
- **Învățarea celor trei principii ale programării orientate spre obiecte**
- **Crearea, compilarea și execuția programelor C#**
- **Utilizarea variabilelor**
- **Lucrul cu instrucțiuni `if` și `for`**
- **Utilizarea blocurilor de cod**
- **Cunoașterea cuvintelor cheie din C#**

Căutarea unui limbaj de programare perfect datează de la începuturile afirmării programării ca disciplină. În această căutare, C# a devenit purtătorul actual al stindardului. Creat de firma Microsoft ca un instrument de dezvoltare pentru arhitectura .NET, C# combină facilități testate de-a lungul timpului cu inovații de ultim moment. Limbajul oferă o modalitate facilă și eficientă de a scrie programe pentru mediul profesional modern de dezvoltare, care cuprinde sistemul Windows, Internetul, componentele software etc. Pe parcursul dezvoltării sale, C# a redefinit peisajul în programare. În cursul lucrării, veți învăța să programați utilizând acest limbaj.

Scopul acestui modul este de a prezenta limbajul C#, incluzând forțele care au condus la crearea acestuia, filozofia proiectării sale și câteva dintre cele mai importante facilități pe care le oferă. De departe, cea mai dificilă problemă legată de învățarea unui limbaj de programare o constituie faptul că nici un element nu este izolat. Componentele unui limbaj de programare lucrează împreună. Această relaționare face dificilă prezentarea unui aspect al limbajului C# fără a implica altele. Pentru a depăși această problemă, acest modul oferă o descriere scurtă a câtorva facilități oferite de C#, printre care forma generală a unui program C#, câteva instrucțiuni de control esențiale și operatorii limbajului. Modulul nu intră în foarte multe detalii, concentrându-se mai degrabă asupra conceptelor generale comune tuturor programelor C#.

Arborele genealogic al limbajului C#

Limbajele de programare nu există într-un spaţiu vid. Dimpotrivă, ele se înrudesc unele cu altele, fiecare limbaj nou apărut fiind influenţat într-un fel sau altul de cele existente până atunci. Într-un proces similar cu polenizarea încrucişată, facilităţi existente într-un limbaj sunt adaptate într-un altul, inovaţiile sunt integrate în contexte existente, sau construcţii mai vechi sunt eliminate. În acest mod, limbajele evoluează, iar ştiinţa programării se dezvoltă. Nici C# nu face excepţie de la această regulă.

C# este moştenitorul unei averi bogate în domeniul programării. Limbajul derivă direct din două dintre cele mai de succes limbaje de programare pe plan mondial: C şi C++. De asemenea, C# este o rudă apropiată a unui alt limbaj de succes: Java. Înţelegerea naturii acestor relaţii este de mare importanţă pentru înţelegerea limbajului C#. Din acest motiv, vom începe prezentarea limbajului C# prin plasarea lui în contextul istoric determinat de aceste trei limbaje.

C: începutul erei moderne a programării

Crearea limbajului C marchează începutul epocii moderne în programare. Limbajul C a fost inventat de către Dennis Ritchie în anii '70 pe un calculator DEC PDP-11 care rula un sistem de operare UNIX. În vreme ce unele limbaje anterioare, dintre care cel mai important era Pascal, înregistrau un succes semnificativ, limbajul C are meritul de a fi definit paradigma care încă trasează cursul programării şi în ziua de azi.

Limbajul C s-a dezvoltat în urma revoluţiei *programării structurate* din anii '60. Înainte de programarea structurată, programele mari erau foarte greu de scris din cauza logicii, care tindea să degenereze în ceea ce este cunoscut sub numele de „cod spaghetti", o masă încâlcită de salturi, apeluri şi reveniri foarte dificil de urmărit. Limbajele structurate au rezolvat această problemă adăugând structuri de control bine definite, subrutine cu variabile locale şi alte îmbunătăţiri. Utilizând limbajele structurate, scrierea programelor moderat de mari a devenit posibilă.

Chiar dacă la acea vreme au existat şi alte limbaje structurate, C a fost primul care a combinat puterea, eleganţa şi expresivitatea. Sintaxa concisă dar uşor de utilizat, cuplată cu ideea că programatorul (şi nu limbajul) conduce ostilităţile a câştigat mulţi adepţi. Poate părea greu de înţeles privind din perspectiva de acum, dar limbajul C reprezenta o gură de aer proaspăt pe care programatorii o aşteptau de mult timp. Ca rezultat, limbajul C a devenit cel mai răspândit limbaj de programare structurată în anii '80.

Venerabilul C are însă şi limitele sale. Una dintre cele mai neplăcute este incapacitatea sa de a lucra cu programe mari. Limbajul C ridică o barieră ori de câte ori un proiect atinge o anumită dimensiune. Peste acest prag, programele scrise în C devin greu de înţeles şi întreţinut. Valoarea precisă a pragului depinde de program, de programator şi de instrumentele disponibile, dar este posibil să se situeze chiar în jurul a 5000 de linii de cod.

Crearea programării orientate spre obiecte şi C++

La finele anilor '70, dimensiunile multor programe erau aproape de limitele impuse de metodologia programării structurate şi de limbajul C respectiv. Pentru a rezolva această problemă, o nouă modalitate de programare a început să apară. Această metodă poartă numele de programare orientată spre obiecte (POO pe scurt). Utilizând POO, un singur programator poate gestiona programe mult mai mari. Neajunsul era faptul că limbajul C, cel mai popular în acea perioadă, nu permitea programarea orientată spre obiecte. Dorinţa de a exista o versiune de C orientată spre obiecte a condus într-un final la crearea limbajului C++.

Limbajul C++ a fost inventat de către Bjarne Stroustrup începând din 1979, la Laboratoarele Bell din Murray Hill, New Jersey. El a denumit iniţial noul limbaj „C cu clase". În 1983, numele a fost însă modificat în „C++". Limbajul C++ include toate facilităţile din C. Prin urmare, C-ul a fost fundamentul peste care a fost construit limbajul C++. Majoritatea completărilor efectuate de Stroustrup asupra C-ului aveau rolul de a permite programarea orientată spre obiecte. În esenţă, C++ reprezintă versiunea orientată spre obiecte a limbajului C. Construind pe fundamentul C-ului, Stroustrup a deschis calea unei migrări lente către POO. Programatorii în C nu mai aveau de învăţat un limbaj cu totul nou, ci era suficient să-şi însuşească câteva noi facilităţi pentru a culege beneficiile aduse de metodologia orientată spre obiecte.

În anii '80, limbajul C++ a stat în linia a doua în majoritatea timpului, suferind în acest timp dezvoltări masive. La începutul anilor '90, C++ era pregătit pentru utilizarea pe scară largă, popularitatea sa înregistrând o creştere explozivă. Până la finele deceniului, C++ a devenit cel mai răspândit limbaj de programare. În prezent, C++ continuă să fie principalul limbaj utilizat la dezvoltarea de programe pentru sisteme nedistribuite.

Este foarte important să înţelegem că inventarea limbajului C++ nu a fost doar o încercare de a crea un nou limbaj de programare, ci o dezvoltare adusă unui limbaj care avea deja succes. Această abordare a dezvoltării limbajelor – începând de la un limbaj existent şi construind peste acesta – a stabilit o tendinţă care continuă şi în prezent.

Internetul şi apariţia limbajului Java

Următorul pas major în evoluţia limbajelor de programare l-a reprezentat Java. Lucrul la acest limbaj, denumit iniţial Oak, a fost demarat în 1991 la firma Sun Microsystems. Principalul artizan al proiectării limbajului Java a fost James Gosling.

O contribuţie importantă au adus şi Patrick Naughton, Chris Warth, Ed Frank şi Mike Sheridan.

Java este un limbaj structurat şi orientat spre obiecte, cu o sintaxă şi o filozofie derivate din C++. Aspectele novatoare aduse de Java nu erau orientate atât spre progresul ştiinţei programării (deşi au fost câteva astfel de aspecte), cât spre modificările mediului de programare. Înainte de explozia Internetului, majoritatea programelor erau scrise, compilate şi destinate utilizării pe un anumit procesor şi sub un anumit sistem de operare. Chiar dacă programatorilor le-a convenit întotdeauna să-şi reutilizeze codul, posibilitatea portării uşoare a unui program dintr-un mediu în altul a fost amânată din cauza unor probleme mai presante. După apariţia Internetului însă, la care sunt conectate sisteme cu procesoare şi sisteme de operare diferite, vechea problemă a portabilităţii a devenit substanţial mai importantă. Pentru rezolvarea problemei portabilităţii a fost necesar un nou limbaj, şi acesta a fost Java.

Chiar dacă singurul aspect esenţial în Java (şi totodată motivul pentru acceptarea sa rapidă) este posibilitatea de a crea cod portabil pe platforme diferite, este interesant să observăm că impulsul iniţial pentru Java nu a fost Internetul, ci necesitatea unui limbaj independent de platformă care să poată fi utilizat la crearea programelor pentru microcontrolere. În 1993, a devenit clar că soluţiile găsite pentru asigurarea portabilităţii în cazul dezvoltării de cod pentru microcontrolere pot fi aplicate şi atunci când se scriu programe pentru Internet. Să ne amintim că Internetul este un univers vast şi distribuit, în care coexistă multe tipuri diferite de calculatoare. Aceleaşi tehnici care au rezolvat problema portabilităţii la o scară restrânsă au putut fi aplicate pe o scară largă în cazul Internetului.

Java a realizat portabilitatea prin transformarea codului sursă al programului într-un limbaj intermediar numit *bytecode*. Acest format intermediar este apoi executat de Maşina Virtuală Java (MVJ). În consecinţă, programele Java pot rula în orice mediu în care este disponibilă o MVJ. Mai mult, deoarece MVJ este relativ uşor de implementat, aceasta a fost imediat disponibilă pentru un număr mare de medii.

Utilizarea formatului bytecode în Java era radical diferită atât faţă de C, cât şi de C++, care mai întotdeauna se compilau până la cod maşină executabil. Codul maşină este legat de un anumit procesor şi de un anumit sistem de operare. În consecinţă, când se dorea rularea unui program C/C++ pe un sistem diferit, acesta trebuia recompilat, generându-se cod maşină specific mediului respectiv. Pentru a crea un program C/C++ care să poată rula într-o varietate de medii, erau necesare mai multe versiuni executabile ale programului. Aceasta nu era numai nepractic, ci şi foarte costisitor. Soluţia de utilizare a unui limbaj intermediar în Java era în acelaşi timp elegantă şi eficientă din punct de vedere al costului. Era totodată o soluţie pe care C# o va adapta la propriile scopuri.

După cum am mai spus, Java derivă din C şi C++. Sintaxa limbajului se bazează pe cea din C, în timp ce modelul obiectual a evoluat din C++. Deşi codul Java nu este compatibil în nici o direcţie cu C sau C++, sintaxa limbajului este suficient de apropiată pentru ca marea masă a programatorilor în C/C++ să poată migra spre Java cu un efort minim. Mai mult, datorită faptului că Java a construit deasupra unei paradigme existente, pe care totodată a şi îmbunătăţit-o, Gosling şi ceilalţi şi-au putut concentra atenţia asupra facilităţilor noi. La fel cum Stroustrup nu a „reinventat roata" atunci când a creat C++, nici Gosling nu a fost nevoit să creeze un limbaj cu totul nou când a dezvoltat Java. În plus, după crearea limbajului Java, C şi C++ au devenit baza peste care se construiesc noile limbaje de programare.

Crearea limbajului C#

Chiar dacă Java a rezolvat cu succes multe din problemele legate de portabilitate în mediul Internet, există încă facilităţi care îi lipsesc. Una dintre acestea este *interoperabilitatea limbajelor diferite*, cunoscută şi sub numele de *programare în limbaj mixt*. Aceasta reprezintă posibilitatea codului scris într-un anumit limbaj de a lucra în mod natural împreună cu codul scris în alt limbaj. Interoperabilitatea limbajelor diferite este esenţială la crearea sistemelor software distribuite de dimensiuni mari. Este de asemenea oportună la programarea componentelor software, deoarece componentele cele mai valoroase sunt acelea care pot fi utilizate în cea mai largă varietate de limbaje de programare şi în cel mai mare număr de medii de operare diferite.

O altă facilitate care nu există în Java este integrarea deplină cu platforma Windows. Deşi programele Java pot fi executate într-un mediu Windows (presupunând că Maşina Virtuală Java a fost instalată), Java şi Windows nu sunt rude apropiate. Cum Windows este sistemul de operare cel mai răspândit în lume, lipsa suportului direct pentru Windows este un neajuns important al limbajului Java.

Pentru a răspunde acestor nevoi şi altora, Microsoft a dezvoltat limbajul C#. C# a fost creat de Microsoft la finele anilor '90 ca parte a ansamblului strategiei .NET, dezvoltată tot de Microsoft. Prima lansare a versiunii alfa a avut loc la mijlocul anului 2000. Arhitectul şef al construcţiei C# este Anders Hejlsberg.

C# este direct înrudit cu C, C++ şi Java. Aceasta nu este o pură întâmplare. Cele trei limbaje sunt printre cele mai răspândite – şi totodată printre cele mai populare – limbaje de programare din lume. În plus, aproape toţi programatorii profesionişti din ziua de azi cunosc C şi C++, iar majoritatea cunosc şi Java. Construind limbajul C# peste un fundament solid şi bine cunoscut, calea de migrare din aceste limbaje către C# a fost considerabil netezită. Cum Hejlsberg nu avea nevoie (şi nici nu era de dorit) să „reinventeze roata", el a avut toată libertatea de a se concentra asupra îmbunătăţirilor şi inovaţiilor specifice.

Arborele genealogic pentru C# este prezentat în figura 1-1. „Bunicul" limbajului C# este C-ul. De la C, C# moşteneşte sintaxa, multe dintre cuvintele cheie şi

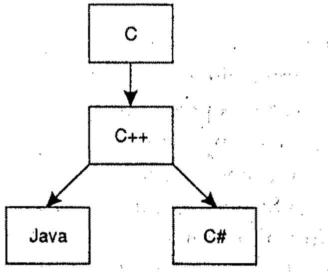


Figura 1-1 Arborele genealogic C#

operatorii. C# construieşte peste modelul de obiecte definit în C++, îmbunătăţindu-l totodată. Dacă ştiţi să programaţi în C sau C++, vă veţi simţi familiar în C#.

Relaţia dintre C# şi Java este ceva mai complicată. După cum am explicat, Java derivă la rândul său din C şi C++. Java moşteneşte de asemenea sintaxa C/C++ şi modelul obiectual. Ca şi Java, C# a fost proiectat pentru a produce cod portabil. Limbajul C# nu derivă însă din Java. Între C# şi Java există o relaţie similară celei dintre „veri", ele având un strămoş comun, dar deosebindu-se prin multe caracteristici importante. Vestea bună este că dacă ştiţi să programaţi în Java, atunci multe dintre conceptele din C# vă vor fi familiare. Reciproc, dacă în viitor veţi fi nevoit să învăţaţi Java, multe dintre lucrurile învăţate despre C# vă vor fi de ajutor. Limbajul C# conţine multe facilităţi novatoare pe care le vom examina în detaliu pe parcursul cărţii, dar unele dintre cele mai importante se referă la suportul încorporat pentru componente software. De fapt, C# a fost caracterizat ca fiind un limbaj orientat spre componente, deoarece conţine un suport complet pentru dezvoltarea de componente software. De exemplu, C# dispune de facilităţi care implementează direct elementele care alcătuiesc componentele, cum ar fi proprietăţile, metodele şi evenimentele. Cea mai importantă facilitate „orientată spre componente" de care dispune C# este probabil posibilitatea de lucru într-un mediu cu limbaj mixt.

Relaţia dintre C# şi arhitectura .NET

Chiar dacă C# este un limbaj de programare ce poate fi studiat separat, are o legătură deosebită cu mediul său de rulare, arhitectura .NET. Şi aceasta din două motive. Mai întâi, C# a fost iniţial dezvoltat de Microsoft pentru crearea codului pentru arhitectura .NET. În al doilea rând, bibliotecile utilizate de C# sunt cele definite de arhitectura .NET. În concluzie, chiar dacă este posibil să separăm limbajul C# de mediul .NET, acestea sunt deocamdată foarte apropiate. Din acest motiv, este bine să avem în linii mari o idee despre arhitectura .NET şi de ce este aceasta atât de importantă pentru C#.

Ce este arhitectura .NET?

Pe scurt, arhitectura .NET defineşte un mediu care permite dezvoltarea şi execuţia aplicaţiilor independente de platformă. Aceasta permite amestecarea diferitelor limbaje de programare şi oferă facilităţi de securitate şi portabilitate a programelor şi

un mediu de programare comun pentru platformele Windows. Este important să menţionăm că arhitectura .NET nu se limitează la Windows (deşi acesta este singurul mediu disponibil deocamdată), ceea ce înseamnă că programele scrise pentru ea pot deveni portabile în medii non-Windows în viitor.

Legat de C#, arhitectura .NET defineşte două entităţi foarte importante. Prima dintre acestea este *motorul comun de programare*. Acesta este sistemul care se ocupă de execuţia programelor dumneavoastră. Pe lângă alte avantaje, motorul comun de programare este partea arhitecturii .NET care permite programelor să fie portabile, asigură programarea în limbaj mixt şi securitatea.

Cea de-a doua entitate este *biblioteca de clase* .NET. Prin intermediul acestei biblioteci, programul dumneavoastră are acces la mediul de rulare. De exemplu, atunci când efectuaţi o operaţie de intrare-ieşire, cum ar fi afişarea unui text pe ecran, utilizaţi de fapt biblioteca de clase .NET pentru aceasta. Dacă sunteţi începător în programare, termenul de *clasă* vă este nou. Deşi va fi explicat în detaliu ceva mai târziu, termenul de clasă desemnează în principiu o construcţie orientată spre obiecte care facilitează organizarea programelor. Atât timp cât programul dumneavoastră se limitează la facilităţile definite în biblioteca de clase .NET, poate rula oriunde motorul de rulare .NET este disponibil. Cum C# utilizează automat biblioteca de clase .NET, programele C# sunt automat portabile în orice mediu .NET.

Operarea motorului comun de programare

Motorul comun de programare se ocupă de execuţia codului .NET. Prezentăm în continuare principiul de funcţionare al acestuia. Atunci când compilaţi un program C#, rezultatul compilării nu este cod executabil. Acesta este înlocuit de un fişier care conţine un tip aparte de pseudocod numit limbaj intermediar Microsoft, sau MSIL pe scurt (de la MicroSoft Intermediate Language). Limbajul MSIL defineşte un set de instrucţiuni portabile care sunt independente de orice tip de procesor. În esenţă, MSIL defineşte un limbaj de asamblare portabil. Un alt aspect este faptul că, deşi MSIL reprezintă un concept similar cu bytecode-ul din Java, cele două nu sunt unul şi acelaşi lucru.

Motorul comun de programare are misiunea de a transforma codul intermediar în cod executabil atunci când se rulează programul. În concluzie, orice program compilat până în format MSIL poate rula în orice mediu pentru care motorul comun de programare este implementat. Aceasta este o parte a modului în care arhitectura .NET asigură portabilitatea.

Limbajul intermediar Microsoft este transformat în cod executabil utilizând un compilator *JIT*. JIT este prescurtarea de la „just in time" (exact la timp). Procesul de conversie decurge astfel: atunci când un program .NET este executat, motorul comun de programare activează compilatorul JIT. Compilatorul JIT converteşte

MSIL în cod nativ la cerere, pe măsură ce fiecare parte a programului este necesară. În acest fel, programul C# se execută de fapt sub formă de cod nativ, chiar dacă inițial este compilat până la formatul MSIL. Aceasta înseamnă că programul rulează aproape la fel de rapid ca și când ar fi fost compilat de la început până la cod nativ, dar câștigă avantajul portabilității, oferit de MSIL.

Pe lângă MSIL, există și un alt rezultat al compilării unui program C#: *metadate*. Metadatele descriu datele utilizate de program și permit interacțiunea codului dumneavoastră cu alt cod. Metadatele se găsesc în același fișier ca și MSIL.

Din fericire, atât pentru scopul pe care și-l propune această lucrare, cât și pentru majoritatea activităților de programare, nu este necesar să cunoașteți mai mult despre motorul comun de programare, MSIL sau metadate. C# se ocupă de aceste detalii în locul dumneavoastră.

Cod tratat sau netratat

În general, atunci când scrieți un program C#, creați ceea ce este cunoscut sub numele de *cod tratat*. Codul tratat se execută sub controlul motorului comun de programare, după cum tocmai am văzut. Din cauză că rulează sub controlul motorului comun de programare, codul tratat este supus unor anumite restricții – dar în același timp beneficiază de anumite avantaje. Restricțiile sunt ușor de descris și de îndeplinit: compilatorul trebuie să producă un fișier MSIL destinat motorului de programare (ceea ce în C# se întâmplă) și să utilizeze bibliotecile arhitecturii .NET (condiție de asemenea îndeplinită în C#). Avantajele codului tratat sunt multiple, printre care gestiunea modernă a memoriei, posibilitatea de a amesteca limbajele, o mai bună securitate, posibilitatea controlului versiunilor și o modalitate curată de interacțiune a componentelor software.

Opusul codului tratat este codul *netratat*. Codul netratat nu se execută sub controlul motorului comun de programare. În concluzie, toate programele Windows anterioare creării arhitecturii .NET conțin cod netratat. Codul tratat și cel netratat pot interacționa; prin urmare, faptul că în C# se generează cod tratat nu restricționează posibilitatea acestuia de a opera în conjuncție cu programe anterior existente.

Specificația comună a limbajelor

Deși codul tratat beneficiază de avantajele oferite de motorul comun de programare, atunci când codul dumneavoastră va fi utilizat de alte programe, pentru o utilizabilitate maximă trebuie să respecte *specificația comună a limbajelor* (SCL). SCL descrie un set de caracteristici, cum ar fi tipuri de date, pe care diferite limbaje le au în comun. Conformitatea cu SCL este importantă mai ales la crearea componentelor software care vor fi utilizate în alte limbaje. Chiar dacă pentru scopul acestei lucrări

nu avem nevoie să ne preocupăm de SCL, aceasta trebuie studiată atunci când începeți să programați în scopuri comerciale.

Sfatul expertului

Întrebare: Pentru a rezolva problemele de portabilitate, securitate și programare în limbaj mixt, de ce a fost necesar să se creeze un nou limbaj de programare cum este C#? Nu putea fi adaptat un limbaj cum este C++ pentru lucrul cu arhitectura .NET?

Răspuns: Da, este posibil să se adapteze limbajul C++ pentru a produce cod compatibil cu arhitectura .NET, care să fie rulat de motorul comun de programare. De fapt, Microsoft chiar așa a procedat, adăugând la C++ ceea ce este cunoscut sub numele de *extensii tratate*. Chiar dacă extensiile tratate în C++ fac posibilă portarea codului pe arhitectura .NET, dezvoltarea noilor programe .NET este mult mai ușoară în C#. După cum ne amintim, C# este optimizat pentru mediul .NET și conține câteva facilități încorporate care accelerează mult dezvoltarea programelor .NET.

Exerciții la minut

- Cu ce limbaje se înrudește C#?
- Ce este motorul comun de programare?
- Ce este un compilator JIT?

Programarea orientată spre obiecte

Ideea centrală a limbajului C# este programarea orientată spre obiecte (POO). Metodologia orientată spre obiecte este inseparabilă de C#, toate programele C# fiind cel puțin într-o anumită măsură orientate spre obiecte. Datorită importanței deosebite în C#, este bine să înțelegeți principiile de bază ale POO înainte de a scrie chiar programe simple în C#.

POO reprezintă o modalitate puternică de abordare a activității de programare. Metodologiile de programare s-au schimbat dramatic de la apariția calculatoarelor, în primul rând pentru a ține pasul cu complexitatea din ce în ce mai mare a programelor. De exemplu, pentru primele calculatoare apărute, programarea se efectua introducând instrucțiunile mașină, scrise în binar, de la panoul frontal al calculatorului.

- C# derivă din C și C++ și este „văr" cu Java.
- Motorul comun de programare dirijează execuția programelor .NET.
- Prescurtarea JIT vine de la „just in time". Un compilator JIT convertește codul MSIL în cod nativ pe măsură ce este nevoie, adică la cerere, pe parcursul execuției programului.

Cum programele aveau doar cel mult câteva sute de instrucţiuni, metoda dădea rezultate. Pe măsură ce programele au crescut, s-a inventat limbajul de asamblare, în care un programator putea gestiona programe mai mari, cu o complexitate sporită, utilizând reprezentări simbolice ale instrucţiunilor maşină. Când programele au continuat să crească, s-au introdus limbaje de nivel înalt, cum ar fi FORTRAN şi COBOL, care ofereau programatorilor mai multe instrumente de tratare a complexităţii. Când şi aceste limbaje timpurii au început să-şi atingă limitele, s-a inventat programarea structurată.

Să concluzionăm: la fiecare piatră de hotar în dezvoltarea programării, au fost create noi tehnici şi instrumente pentru a permite programatorilor să rezolve problema complexităţii din ce în ce mai mari. La fiecare pas, noua abordare a preluat ceea ce era mai bun din metodele anterioare, mergând înainte. Aceasta rămâne valabil şi pentru programarea orientată spre obiecte. Înainte de apariţia POO, multe proiecte s-au apropiat (sau chiar au trecut) de punctul din care abordarea structurată nu mai funcţionează. Era necesară o modalitate superioară de tratare a complexităţii, iar soluţia a fost programarea orientată spre obiecte.

Programarea orientată spre obiecte a preluat cele mai bune idei de la programarea structurată, combinându-le cu câteva concepte noi. Rezultatul a fost o modalitate diferită şi totodată mai bună de a organiza un program. La nivelul cel mai general, un program poate fi organizat în două moduri: în jurul codului (a ceea ce se întâmplă în program) sau în jurul datelor (a ceea ce este afectat de execuţia programului). Utilizând numai tehnicile specifice programării structurate, programele sunt de regulă organizate în jurul codului. Această abordare poate fi descrisă prin sintagma „codul acţionează asupra datelor".

Programele orientate spre obiecte funcţionează utilizând cealaltă abordare. Ele sunt organizate în jurul datelor, principiul fundamental fiind „datele controlează accesul la cod". Într-un limbaj orientat spre obiecte, se definesc datele şi rutinele cărora li se permite să acţioneze asupra datelor. În concluzie, un tip de date defineşte în mod exact tipurile de operaţii care sunt permise asupra acelor date.

Pentru a implementa principiile programării orientate spre obiecte, toate limbajele POO, inclusiv C#, au trei caracteristici comune: încapsularea, polimorfismul şi moştenirea. Vom studia separat pe fiecare dintre acestea.

Încapsularea

Încapsularea este un mecanism de programare care combină codul şi datele pe care acesta le manipulează, menţinând integritatea acestora faţă de interferenţa cu lumea exterioară şi utilizarea necorespunzătoare. Într-un limbaj orientat spre obiecte, codul şi datele pot fi legate împreună în aşa fel încât se creează o *cutie neagră*. Datele şi codul se găsesc în interiorul cutiei. Atunci când codul şi datele sunt combinate în

această manieră, se creează un obiect. Cu alte cuvinte, un obiect este un dispozitiv care implementează încapsularea.

În cadrul unui obiect, codul, datele, sau amândouă pot fi *private* sau *publice*. Codul sau datele private sunt cunoscute şi totodată accesibile numai dintr-o altă parte a aceluiaşi obiect. Prin urmare, codul sau datele private nu vor fi accesibile dintr-o parte a programului care există în afara acelui obiect. Atunci când codul sau datele sunt publice, celelalte părţi ale programului le pot utiliza, chiar dacă ele sunt definite în interiorul unui obiect. De regulă, părţile publice ale unui obiect sunt utilizate pentru a furniza o interfaţă controlată cu elementele private ale obiectului.

Unitatea fundamentală de încapsulare în C# o reprezintă *clasa*. O clasă defineşte forma unui obiect. Clasa specifică atât datele, cât şi codul care operează asupra acelor date. Specificaţia unei clase este utilizată de C# pentru a construi *obiecte*. Obiectele reprezintă instanţe ale clasei. În concluzie, o clasă reprezintă de fapt o mulţime de planuri care precizează cum se construiesc obiectele.

Codul şi datele care constituie o clasă sunt denumite *membri* ai clasei. Mai precis, datele definite în cadrul clasei sunt denumite *variabile membru*, sau *variabile instanţă*. Codul care operează asupra datelor este denumit prin termenul de *metode membru* sau pe scurt *metode*. „Metodă" este termenul C# pentru o subrutină. Dacă limbajele C/C++ vă sunt familiare, este util să ştiţi că termenul de *metodă* utilizat de programatorii în C# desemnează ceea ce programatorii în C/C++ numesc *funcţie*. Cum C# este un descendent direct al limbajului C++, termenul de „funcţie" va fi de asemenea utilizat uneori pentru a denumi o metodă în C#.

Polimorfismul

Polimorfismul (derivat de la un termen grecesc cu semnificaţia „cu mai multe forme") este calitatea care permite unei interfeţe să aibă acces la o clasă generică de acţiuni. Un exemplu simplu de polimorfism îl putem întâlni la volanul unui automobil. Volanul (interfaţa) este acelaşi, indiferent de tipul de mecanismul de direcţie utilizat efectiv. Aceasta înseamnă că volanul funcţionează la fel, indiferent dacă maşina are direcţie manuală, servodirecţie sau mecanism cu cremalieră şi pinion. În consecinţă, întoarcerea volanului spre stânga determină maşina să se deplaseze la stânga indiferent de tipul mecanismului de direcţie. Beneficiul interfeţei uniforme este că, desigur, de îndată ce aţi învăţat să manevraţi volanul, puteţi conduce orice tip de maşină.

Acelaşi principiu se poate aplica de asemenea şi în programare. De exemplu, să considerăm o stivă (care este o listă cu disciplina „primul sosit, ultimul servit"). Este posibil ca un program să necesite trei tipuri diferite de stive. O stivă poate fi utilizată pentru valori întregi, o alta pentru valori în virgulă mobilă, iar cea de-a treia pentru caractere. În acest caz, algoritmul care implementează fiecare stivă este acelaşi,

chiar dacă datele memorate diferă. Într-un limbaj care nu este orientat spre obiecte, este necesar să creăm trei seturi distincte de rutine de lucru cu stive, rutinele din fiecare set având nume proprii. Datorită polimorfismului, în C# putem crea un singur set generic de rutine de lucru cu stive, care funcționează în toate cele trei cazuri particulare. În acest fel, de îndată ce știm să utilizăm o stivă, vom putea utiliza orice stivă.

Mai general, conceptul de polimorfism este exprimat adeseori prin sintagma „o singură interfață, mai multe metode". Aceasta înseamnă că este posibil să stabilim o interfață generică pentru un întreg grup de activități înrudite. Polimorfismul ajută la reducerea complexității permițând aceleiași interfețe să fie folosită pentru a specifica o *clasă generică de acțiuni*. Selecția *acțiunii specifice* (adică a metodei) care se aplică în fiecare situație în parte este misiunea compilatorului. Dumneavoastră ca programator nu trebuie să efectuați manual această selecție. Trebuie doar să rețineți și să utilizați interfața generică.

Moștenirea

Moștenirea este procesul prin care un anumit obiect poate dobândi proprietățile altui obiect. Aceasta este important, deoarece implementează conceptul de clasificare ierarhică. Dacă stăm să ne gândim, majoritatea cunoștințelor sunt ușor de însușit tocmai datorită unor clasificări ierarhice (adică de sus în jos). De exemplu, un măr ionatan face parte din categoria *măr*, care este o parte a clasei *fruct*, care este subordonată clasei mai mari *aliment*. În consecință, clasa *aliment* posedă anumite caracteristici (comestibil, nutritiv etc.) care, în mod logic, se transmit și subclasei sale *fruct*. În afară de aceste caracteristici, clasa *fruct* are și altele particulare (zemos, dulce ș.a.m.d), care o disting de alte alimente. Clasa *măr* definește acele însușiri specifice unui măr (crește în pomi, nu este tropical etc.). Un măr ionatan va moșteni prin urmare caracteristicile tuturor claselor precedente, definind numai acele trăsături care îl identifică în mod unic.

Fără a utiliza ierarhii, fiecare obiect ar trebui să-și definească în mod explicit toate caracteristicile. Utilizând moștenirea, un obiect trebuie să definească numai acele trăsături care îl identifică în mod unic în cadrul clasei. Atributele generice pot fi moștenite de la obiectul părinte. În concluzie, mecanismul moștenirii este cel care face posibil ca un obiect să fie o instanță a unui caz mai general.

Exerciții la minut

- Menționați principiile POO.
- Care este unitatea fundamentală de încapsulare în C#?

- Principiile OOP sunt încapsularea, polimorfismul și moștenirea.
- Unitatea fundamentală de încapsulare este clasa.

Sfatul expertului

Întrebare: S-a precizat că programarea orientată spre obiecte reprezintă o modalitate eficientă de a gestiona programe mari. Se pare însă că această manieră determină o reducere substanțială a performanțelor pentru programele relativ mici. Deoarece s-a stabilit că toate programele C# sunt într-o anumită măsură orientate spre obiecte, nu este acesta un handicap pentru programele mai mici?

Răspuns: Nu. După cum veți observa, pentru programele mici, facilitățile orientate spre obiecte din C# sunt aproape transparente. Deși este adevărat că limbajul C# urmărește un model obiectual strict, aveți toată libertatea în alegerea măsurii în care îl întrebuințați. Pentru programele mici, „orientarea spre obiecte" este abia perceptibilă. Pe măsură ce programele dumneavoastră cresc, veți integra din ce în ce mai multe facilități orientate spre obiecte cu un efort minim.

Primul program simplu

Înainte de a intra în mai multe detalii, vom începe cu compilarea și rularea unui scurt program C# prezentat ca exemplu mai jos:

```
/*
   Acesta este un program simplu in C#

   Denumiti acest program Example.cs
*/

using System;

class Example {

  // Orice program in C# incepe cu apelul metodei Main().
  public static void Main() {
    Console.WriteLine("Un program simplu in C#.");
  }
}
```

Există două moduri de a edita, compila și rula un progrtam în C#. În primul rând, puteți utiliza compilatorul în linie de comandă, **csc.exe**. A doua posibilitate este să utilizați mediul de dezvoltare integrată (IDE) Visual C++. Ambele metode sunt prezentate în continuare.

Utilizarea compilatorului C# în linie de comandă csc.exe

Chiar dacă pentru proiectele dumneavoastră comerciale veţi utiliza probabil mediul integrat Visual C++, compilatorul C# în linie de comandă reprezintă modalitatea cea mai simplă de a compila şi rula majoritatea exemplelor prezentate în această carte. Pentru a crea şi rula programe utilizând compilatorul C# în linie de comandă, trebuie să parcurgeţi următorii trei paşi:

1. Introduceţi textul programului.
2. Compilaţi programul.
3. Rulaţi programul.

Introducerea programului

Programele prezentate în această lucrare sunt disponibile pe situl Web al editurii Osborne: **www.osborne.com**. Aveţi însă libertatea să introduceţi programele manual. În acest caz, introduceţi textul programului în calculator utilizând un editor de texte, ca de exemplu Notepad. Reţineţi că trebuie să creaţi fişiere text şi nu fişiere formatate cu procesoarele avansate de texte, deoarece informaţia de formatare dintr-un astfel de fişier nu este recunoscută de către compilatorul de C#. După ce aţi introdus programul, denumiţi fişierul **Example.cs**.

Compilarea programului

Pentru a compila programul, executaţi compilatorul C#, **csc.exe**, precizând numele fişierului sursă în linia de comandă, după cum observaţi mai jos:

```
C:\>csc Example.cs
```

Compilatorul **csc** creează un fişier numit **Example.exe**, care conţine versiunea MSIL a programului. Chiar dacă formatul MSIL nu reprezintă cod executabil, acesta este conţinut tot într-un fişier cu extensia **.exe**. Motorul comun de programare invocă automat compilatorul JIT atunci când încercaţi să executaţi **Example.exe**. Fiţi însă sigur că, dacă veţi încerca să executaţi **Example.exe** (sau orice alt fişier cu extensia **.exe** care conţine MSIL) pe un calculator pe care arhitectura .NET nu este instalată, programul nu se va executa, deoarece motorul comun de programare lipseşte.

Observaţie

Înainte de a rula **csc.exe**, este necesar să rulaţi fişierul de comenzi **vcvars32.bat**, care se găseşte de regulă în directorul *\Program Files\Microsoft Visual Studio.NET\Vc7\Bin*. O altă posibilitate este să activaţi o sesiune de lucru în linie de comandă gata iniţializată pentru C#, selectând opţiunea *Visual Studio.NET Command Prompt* din lista de instrumente prezentată în cadrul intrării *Microsoft Visual Studio.NET 7.0* din meniul *Start|Programs* din bara de taskuri.

Rularea programului

Pentru a rula programul, este suficient să-i tastaţi numele în linia de comandă, după cum observaţi şi mai jos:

```
C:\>Example
```

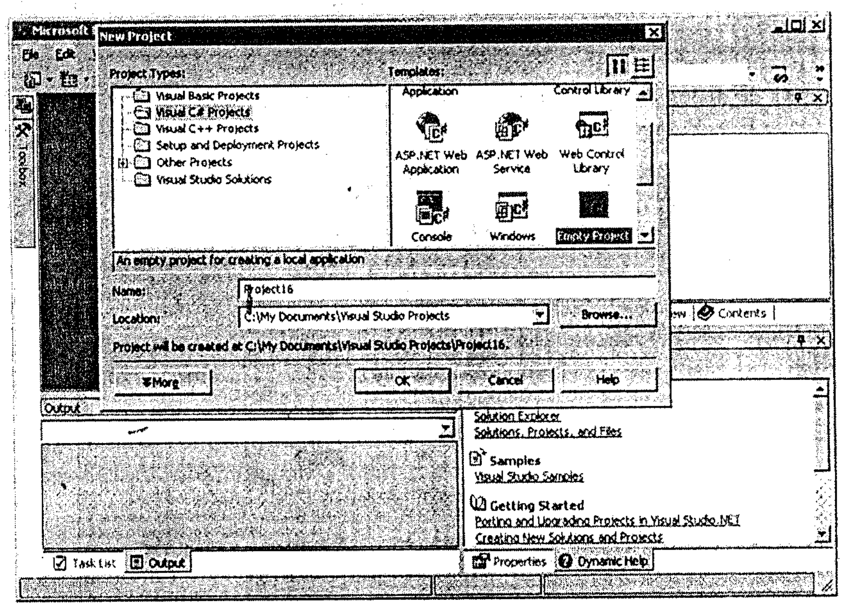
La rulare, programul va afişa următoarea linie:

```
Un program simplu in C#.
```

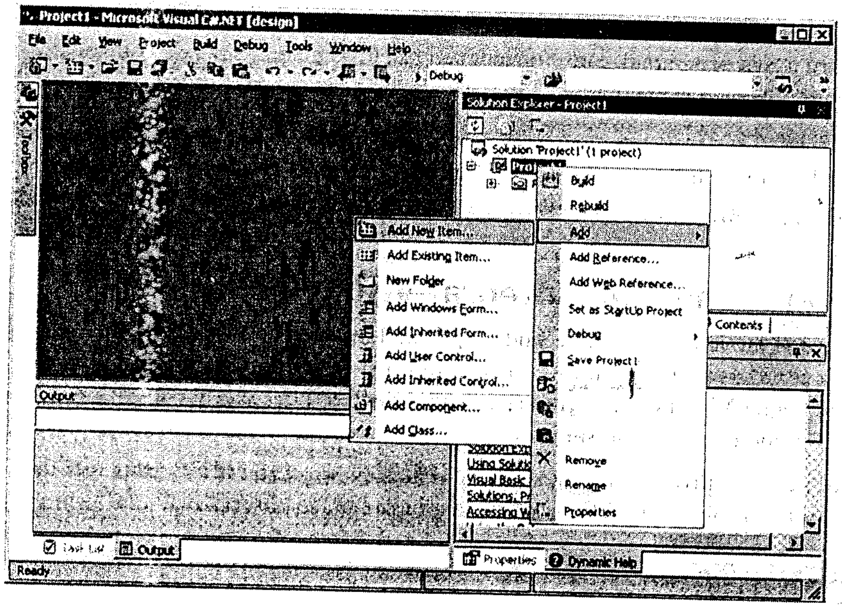
Utilizarea mediului integrat Visual C++

Începând cu versiunea Visual Studio 7, mediul integrat Visual C++ poate compila şi programe C#. Pentru a edita, compila şi rula un program C# utilizând mediul integrat Visual C++ 7, trebuie să parcurgeţi paşii descrişi în continuare. (Dacă dispuneţi de o altă versiune de Visual C++, aceştia se pot modifica.)

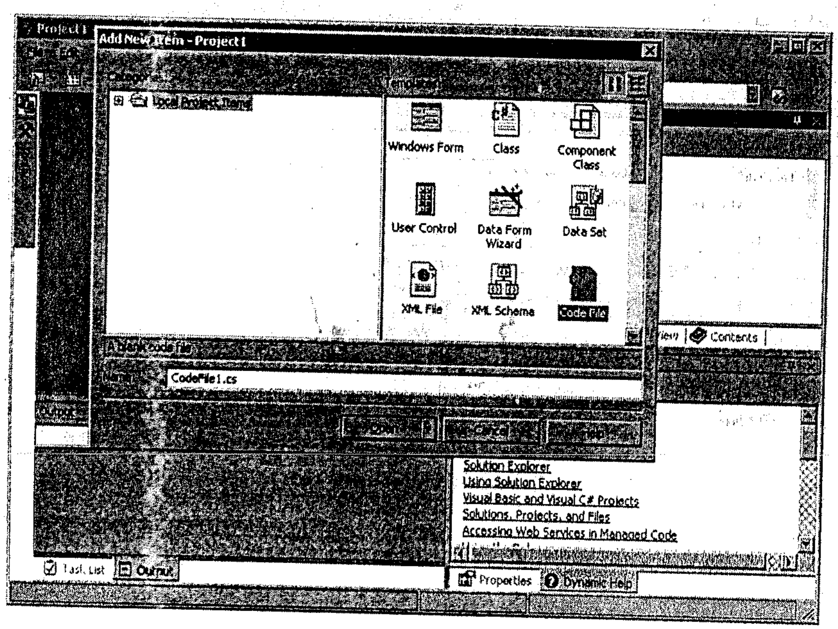
1. Creaţi un nou proiect C# vid, selectând File | New | Project. Selectaţi apoi Visual C# Projects, apoi Empty Project, după cum se observă mai jos:



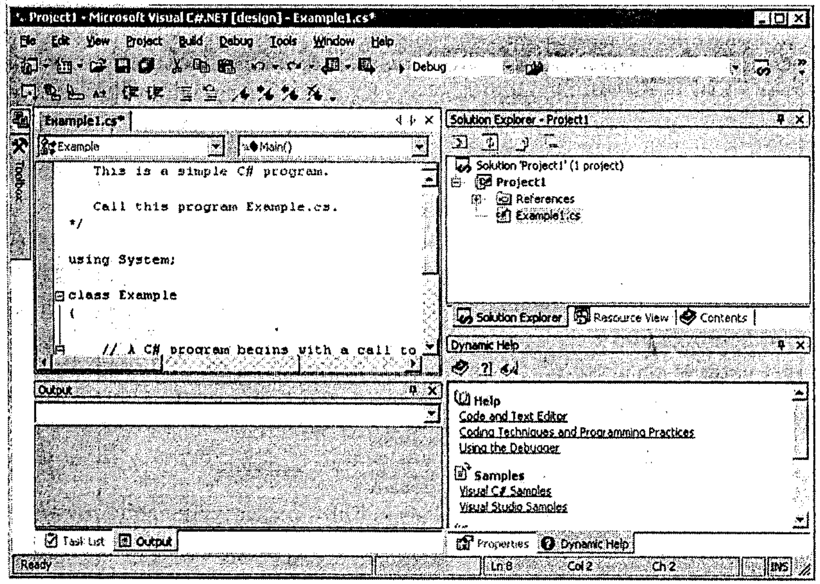
2. După ce aţi creat proiectul, executaţi clic dreapta pe numele proiectului în fereastra Solution. Utilizând meniul contextual apărut, selectaţi Add. Selectaţi apoi Add New Item. Ecranul va arăta ca mai jos:



3. În caseta de dialog Add New Item, selectaţi Local Project Items. În fine, selectaţi C# Code File. Ecranul va arăta astfel:



4. Introduceţi textul programului şi salvaţi fişierul, utilizând numele `Example.cs`. (Reţineţi că puteţi lua programele din această carte de pe situl www.osborne.com.) După ce aţi terminat, ecranul va arăta astfel:



5. Compilaţi programul selectând Build din meniul Build.
6. Rulaţi programul selectând Start Without Debugging din meniul Debug.

La rularea programului, va apărea fereastra prezentată în figura 1-2.

Pentru compilarea şi rularea exemplelor din această carte, nu este nevoie să creaţi un proiect separat pentru fiecare. Puteţi utiliza acelaşi proiect C#. Este suficient să ştergeţi fişierul curent, adăugând noul fişier. Recompilaţi şi rulaţi apoi programul.

După cum am mai precizat, pentru programele scurte prezentate în prima parte a cărţii, utilizarea compilatorului în linia de comandă `csc` este o metodă mult mai simplă. Desigur că aveţi deplină libertate de alegere.

Primul exemplu discutat linie cu linie

Chiar dacă `Example.cs` este foarte scurt, el conţine câteva facilităţi esenţiale, comune tuturor programelor C#. Să examinăm mai îndeaproape fiecare parte a programului, începând cu numele său.

Numele unui program C# poate fi ales la întâmplare. Spre deosebire de alte limbaje de programare (dintre care cel mai important este Java) la care numele fişierului care conţine programul contează foarte mult, în C# lucrurile nu stau aşa.

Vi s-a indicat să denumiţi fişierul care conţine exemplul **Example.cs** pentru ca instrucţiunile de compilare şi rulare să rămână valabile, dar din punctul de vedere al limbajului C#, era posibil să folosiţi pentru fişier orice alt nume. De exemplu, programul din exemplul anterior putea fi denumit **Sample.cs**, **Test.cs** sau chiar **X.cs**.

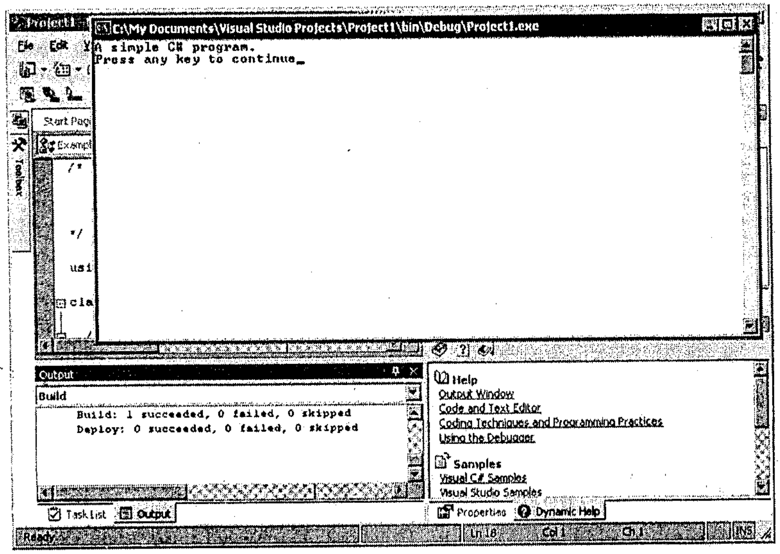


Figura 1-2 Rularea programului **Example.exe** din mediul integrat Visual C++.

Prin convenţie, programele C# utilizează extensia **.cs**; aceasta este o regulă pe care trebuie să o respectaţi. De asemenea, mulţi programatori denumesc fişierul după numele clasei principale definite în cadrul acestuia. Iată de ce am ales în acest caz numele **Example.cs**. Cum numele programelor C# sunt arbitrare, ele nu vor fi specificate pentru majoritatea exemplelor din cadrul lucrării. Utilizaţi nume la libera dumneavoastră alegere.

Programul începe cu următoarele linii:

```
/* Acesta este un program simplu in C#
   Denumiti acest program Example.cs
*/
```

Acesta este un *comentariu*. Ca şi majoritatea celorlalte limbaje de programare, şi C# vă permite să vă introduceţi propriile remarci în fişierul sursă al programului. Conţinutul comentariilor este ignorat de către compilator. Un comentariu prezintă sau explică modul de operare al programului pentru oricine citeşte codul sursă. În acest caz, comentariul descrie programul şi vă aminteşte să denumiţi fişierul sursă

Example.cs. Desigur că, în aplicaţiile reale, comentariile explică în general modul de funcţionare al unei părţi a programului sau rolul unei anumite facilităţi.

În C# putem întâlni trei stiluri de comentarii. Cel care se află în partea de sus a programului este numit *comentariu multilinie*. Acest tip de comentariu trebuie să înceapă cu /* şi să se termine cu */. Orice se găseşte între aceste două simboluri specifice comentariilor este ignorat de către compilator. După cum şi numele arată, comentariile multilinie se pot întinde pe mai multe linii.

Următoarea linie a programului este:

```
using System;
```

Această linie arată că programul utilizează spaţiul de nume **System**. În C#, un *spaţiu de nume* defineşte o regiune declarativă. Chiar dacă vom discuta despre spaţiile de nume în detaliu ceva mai târziu, precizăm acum că ele oferă o modalitate de a menţine un set de nume separat faţă de altele. Spaţiul de nume utilizat în program este **System**, care este spaţiul rezervat elementelor asociate cu biblioteca de clase a arhitecturii .NET, bibliotecă utilizată de către C#. Cuvântul cheie **using** precizează efectiv faptul că programul foloseşte numele din spaţiul de nume dat.

Următoarea linie de cod din program este prezentată mai jos:

```
class Example {
```

Această linie utilizează cuvântul cheie **class** pentru a declara faptul că se defineşte o nouă clasă. După cum am menţionat, clasa reprezintă unitatea fundamentală de încapsulare în C#. **Example** este aici numele clasei. Definiţia clasei începe cu o acoladă deschisă ({) şi se termină cu o acoladă închisă (}). Elementele cuprinse între cele două acolade sunt membrii clasei. Pentru moment, nu vă preocupaţi prea mult de detaliile referitoare la clase; reţineţi însă că în C#, întreaga activitate a unui program are loc în cadrul unei clase. Acesta este unul din motivele pentru care toate programele C# sunt (cel puţin în mică măsură) orientate spre obiecte.

Următoarea linie din program este un *comentariu pe o singură linie*, prezentat mai jos:

```
// Orice program in C# incepe cu apelul metodei Main().
```

Acesta este al doilea tip de comentarii cunoscut de C#. *Un comentariu pe o singură linie* începe cu // şi se termină la sfârşitul liniei. De obicei, programatorii întrebuinţează comentariile multilinie pentru remarci mai lungi, iar pe cele pe o singură linie la descrieri scurte, aferente unei singure linii.

Următoarea linie de cod este cea prezentată în continuare:

```
public static void Main() {
```

Această linie reprezintă începutul metodei **Main()**. După cum am văzut anterior, în C# subrutinele sunt denumite metode. După cum ne indică şi comentariul care precede metoda, aceasta este linia cu care începe execuţia programului. Toate aplicaţiile C# îşi încep execuţia apelând **Main()**. (Aceasta este similar cu programele

C/C++, care îşi încep execuţia cu funcţia `main()`.) Sensul complet al fiecărui element de pe această linie nu poate fi explicat acum, deoarece presupune o înţelegere detaliată a altor câteva facilităţi din C#. Cum însă multe dintre exemplele din această carte vor utiliza această linie de cod, vom arunca o scurtă privire asupra ei acum.

Cuvântul cheie `public` este un specificator de *acces*. Specificatorii de acces determină modul în care alte părţi ale programului au acces la membrii unei clase. Atunci când un membru al unei clase este precedat de `public`, acel membru poate fi utilizat şi de cod aflat în afara clasei în care a fost declarat. (Opusul lui `public` este `private`, care împiedică un membru să fie utilizat de cod definit în afara clasei sale.) În acest caz, metoda `Main()` este declarată ca `public`, deoarece va fi apelată de cod aflat în afara clasei sale (mai precis în sistemul de operare) atunci când programul este lansat în execuţie.

Observaţie

La momentul apariţiei acestei lucrări, C# nu cere ca metoda `Main()` să fie declarată în mod obligatoriu `public`. Acesta este însă modul în care multe dintre exemplele care se găsesc în Visual Studio.NET o declară. Este de asemenea modul preferat de către mulţi programatori de C#. Din aceste motive, şi această lucrare va declara metoda `Main()` ca `public`. Nu este însă cazul să fiţi surprinşi dacă întâlniţi o declaraţie care diferă de aceasta.

Cuvântul cheie `static` permite metodei `Main()` să fie apelată înaintea creării unui obiect din clasa sa. Aceasta este necesar, deoarece `Main()` este apelată la lansarea programului. Cuvântul cheie `void` informează compilatorul că metoda `Main()` nu întoarce nici o valoare. După cum veţi vedea, metodele pot să şi întoarcă valori. Parantezele vide care urmează după `Main` indică faptul că metoda `Main()` nu primeşte nici o informaţie. După cum se va vedea, este posibil să transmitem anumite informaţii metodei `Main()` sau oricărei alte metode. Ultimul caracter de pe linie este acolada deschisă {. Aceasta semnalează începutul corpului metodei `Main()`. Întreg codul care intră în componenţa unei metode apare între acolada deschisă de la începutul metodei şi cea închisă de la sfârşit.

Prezentăm în continuare următoarea linie de cod. Observaţi că aceasta apare în interiorul lui `Main()`.

```
Console.WriteLine("Un program simplu in C#");
```

Această linie afişează şirul de caractere „Un program simplu in C#", urmat de o nouă linie pe ecran. Afişarea este de fapt realizată de către metoda predefinită `WriteLine()`. În acest caz, `WriteLine()` afişează şirul de caractere care îi este transmis. Informaţia transmisă unei metode poartă numele de *parametru*. Pe lângă şiruri de caractere, `WriteLine()` mai poate fi utilizată şi la afişarea altor tipuri de informaţii. Linia începe cu `Console`, care este numele unei clase predefinite care implementează operaţii de intrare-ieşire prin intermediul consolei. Asociind `Console` şi `WriteLine()`, îi

comunicăm compilatorului că `WriteLine()` este un membru al clasei `Console`. Faptul că limbajul C# utilizează o clasă pentru a defini ieşirea la consolă este încă o dovadă a naturii sale orientate spre obiecte.

Observaţi că instrucţiunea `WriteLine()` se termină cu punct şi virgulă, la fel ca şi instrucţiunea anterioară `using System`. Toate instrucţiunile în C# se termină cu punct şi virgulă. Motivul pentru care alte câteva linii din program nu se termină cu punct şi virgulă este că acelea nu reprezintă din punct de vedere tehnic instrucţiuni.

Prima acoladă închisă } din program încheie metoda `Main()`, iar ultima încheie definiţia clasei `Example`.

Sfatul expertului

Întrebare: Aţi precizat că în C# se pot întâlni trei tipuri de comentarii, dar aţi menţionat numai două dintre acestea. Care este al treilea?

Răspuns: Al treilea tip de comentarii permis în C# este *comentariul XML*. Un comentariu XML utilizează etichete XML pentru a vă ajuta să creaţi cod care îşi generează automat documentaţia.

O ultimă observaţie: limbajul C# este sensibil la tipurile de literă. Ignorarea acestui aspect poate fi cauza unor probleme serioase. De exemplu, dacă din întâmplare în loc de `Main` tastaţi `main`, sau `writeline` în loc de `WriteLine`, programul anterior va fi incorect. Mai mult, chiar dacă o clasă care nu conţine o metodă `Main()` este compilată de către compilatorul de C#, acesta nu are nici o modalitate de a o executa. În consecinţă, dacă scrieţi greşit `Main`, compilatorul va compila programul. Va apărea însă un mesaj de eroare care precizează că `Example.exe` nu are definit nici un punct de intrare.

Exerciţii la minut

- Unde începe execuţia unui program C#?
- Ce face instrucţiunea `Console.WriteLine()`?
- Cum se numeşte compilatorul în linie de comandă pentru C#?

- Execuţia unui program C# începe cu metoda `Main()`.
- `Console.WriteLine()` afişează anumite informaţii pe consolă.
- Compilatorul C# în linie de comandă este `csc.exe`.

Tratarea erorilor de sintaxă

Dacă nu aţi încercat încă, introduceţi, compilaţi şi rulaţi programul anterior. După cum poate ştiţi din experienţa anterioară în programare, este destul de uşor să tastăm ceva incorect atunci când introducem codul în calculator. Din fericire, dacă aţi introdus ceva incorect, compilatorul va raporta un mesaj de eroare de sintaxă atunci când va încerca să compileze programul. Compilatorul de C# încearcă să înţeleagă ceva din codul sursă, indiferent ce aţi scris acolo. Din acest motiv, este posibil ca eroarea raportată să nu reflecte întotdeauna cauza reală a problemei. În programul anterior, de exemplu, omisiunea accidentală a acoladei deschise de după **Main()** va genera următoarea secvenţă de erori la compilarea cu compilatorul în linie de comandă **csc**. (Erori similare sunt generate şi la compilarea din mediul integrat.)

```
Example.cs(12, 28): error CS1002: ; expected
Example.cs(13, 22): error CS1519: Invalid token '(' in class,
struct, or interface member declaration
Example.cs(15, 1): error CS1022: Type or namespace definition, or
end-of-file expected
```

În mod evident, primul mesaj de eroare este complet greşit, deoarece ceea ce lipseşte nu este un punct şi virgulă, ci o acoladă. Următoarele două mesaje sunt la fel de bizare.

Scopul acestei discuţii este ca atunci când programul dumneavoastră conţine o eroare de sintaxă, nu trebuie neapărat să luaţi mesajele compilatorului drept bune. Acestea vă pot induce în eroare. Este nevoie în acest caz să interpretaţi încă o dată mesajul pentru a descoperi problema. Examinaţi de asemenea câteva linii de cod aflate imediat înaintea celei în care a fost raportată eroarea. Uneori, o eroare nu este raportată decât la câteva linii după punctul în care ea apare.

O mică modificare

Chiar dacă toate programele din carte o vor utiliza, instrucţiunea:

```
using System;
```

cu care începe primul exemplu nu este, tehnic vorbind, necesară. Utilizarea ei aduce însă un avantaj preţios. Motivul pentru care instrucţiunea nu este necesară este că în C# puteţi întotdeauna utiliza *forma completă* a unui nume, obţinută prin prefixarea cu spaţiul de nume căruia acesta îi aparţine. De exemplu, linia:

```
Console.WriteLine("Un program simplu in C#");
```

se poate rescrie sub forma:

```
System.Console.WriteLine("Un program simplu in C#");
```

În concluzie, codul primului exemplu poate fi rescris după cum urmează:


```
// Aceasta versiune nu contine instructiunea using System
class Example {

  // Orice program in C# incepe cu apelul metodei Main().
  public static void Main() {

    // Aici, Console.WriteLine este in forma completa.
    System.Console.WriteLine("Un program simplu in C#.");   ← Forma completă pentru Console.WriteLine.
  }
}
```

Deoarece este incomodă specificarea spaţiului de nume **System** ori de câte ori utilizăm un membru al său, majoritatea programatorilor în C# includ instrucţiunea **using System** la începutul programelor; la fel se va proceda şi pentru toate programele din această lucrare. Este însă important să înţelegeţi că puteţi completa în mod explicit un nume cu spaţiul de nume căruia îi aparţine, dacă aceasta se impune.

Un al doilea program simplu

Este probabil că nici o altă construcţie dintr-un limbaj de programare nu este mai importantă decât atribuirea unei valori pentru o variabilă. O *variabilă* reprezintă o locaţie de memorie cu nume, căreia îi poate fi atribuită o valoare. Mai mult, valoarea unei variabile se poate modifica pe parcursul execuţiei unui program. Prin urmare, conţinutul unei variabile poate fi modificat, nefiind fixat.

Programul următor creează două variabile, numite **x** şi **y**:

```
// Acest program prezinta utilizarea variabilelor

using System;

class Example2 {
  public static void Main() {
    int x;   // aceasta este declaratia unei variabile   ← declaratia variabilelor
    int y;   // aceasta este declaratia altei variabile

    x = 100;  // se atribuie valoarea 100 lui x   ← se atribuie 100 lui x

    Console.WriteLine("x contine " + x);

    y = x/2;

    Console.Write("y contine x/2: ");
    Console.WriteLine(y);
  }
}
```

La rularea programului, acesta va afişa următoarele rezultate:

```
x contine 100
y contine x/2: 50
```

Acest program introduce câteva concepte noi. Mai întâi, instrucţiunea:

```
int x; // aceasta este declaratia unei variabile
```

declară o variabilă denumită **x**, de tip întreg. În C#, toate variabilele trebuie declarate înainte de a fi utilizate. Mai mult, trebuie precizat şi tipul de valori pe care variabila le poate memora. Acesta este denumit *tipul* variabilei. În acest caz, **x** poate memora valori întregi. În C#, pentru a declara o variabilă de tip întreg, numele acesteia trebuie precedat de cuvântul cheie **int**. În concluzie, instrucţiunea anterioară declară o variabilă numită **x** de tip **int**.

Linia următoare declară o a doua variabilă, numită **y**.

```
int y; // aceasta este declaratia altei variabile
```

Observaţi că formatul utilizat este identic cu al primei declaraţii, singura diferenţă fiind numele variabilei.

În cazul general, pentru a declara o variabilă, se va utiliza o instrucţiune de forma:

tip nume-var;

Aici, *tip* precizează tipul variabilei declarate, iar *nume-var* este numele acesteia. Pe lângă **int**, în C# mai întâlnim şi alte câteva tipuri de date.

Următoarea linie de cod atribuie lui **x** valoarea 100:

```
x = 100; // se atribuie valoarea 100 lui x
```

În C#, operatorul de atribuire este semnul egal. Acesta realizează copierea valorii din partea dreaptă în variabila din stânga.

Următoarea linie de cod afişează valoarea lui **x**, precedată de şirul de caractere „x contine".

```
Console.WriteLine("x contine " + x);
```

În această instrucţiune, semnul plus determină ca valoarea lui **x** să fie afişată după şirul de caractere care o precede. Acest procedeu poate fi generalizat. Utilizând operatorul +, puteţi concatena oricâte elemente doriţi într-o singură instrucţiune **WriteLine()**.

Linia următoare îi atribuie lui y valoarea lui x, împărţită la 2.

```
y = x/2;
```

Această linie realizează împărţirea lui **x** la 2, rezultatul fiind apoi depus în **y**. Prin urmare, după execuţia acestei linii, **y** va conţine valoarea 50. Valoarea lui **x** rămâne nemodificată. Ca şi multe alte limbaje de programare, C# conţine un set complet de operatori aritmetici, printre care şi cei prezentaţi mai jos:

+	Adunare
–	Scădere
*	Înmulţire
/	Împărţire

Iată şi următoarele două linii ale programului:

```
Console.Write("y contine x/2: ");
Console.WriteLine(y);
```

Aici întâlnim două aspecte noi. Mai întâi, metoda predefinită **Write()**, utilizată pentru afişarea şirului de caractere „y contine x/2:". Acest şir *nu* este urmat de o linie nouă. Aceasta înseamnă că, atunci când următorul rezultat va fi generat, acesta va fi afişat începând de pe aceeaşi linie. Metoda **Write()** este similară cu **WriteLine()**, exceptând faptul că nu trece la o linie nouă după fiecare apel. În al doilea rând, în apelul metodei **WriteLine()**, observaţi că **y** este utilizat direct. Atât **Write()**, cât şi **WriteLine()** pot fi utilizate pentru afişarea valorilor având orice tip predefinit din C#.

Încă un aspect legat de declararea variabilelor înainte de a trece mai departe: este posibil să declarăm două sau mai multe variabile utilizând aceeaşi instrucţiune declarativă. Pentru aceasta, este suficient să separăm numele acestora prin virgule. De exemplu, **x** şi **y** puteau fi declarate astfel:

```
int x, y; // ambele sunt declarate intr-o singura instructiune
```

Un alt tip de date

În programul anterior, am utilizat variabile de tipul **int**. Variabilele de tipul **int** pot însă memora numai numere întregi. Nu le putem, deci, utiliza atunci când avem nevoie de elemente fracţionare. De exemplu, o variabilă de tip **int** poate memora valoarea 18, dar nu şi valoarea 18.3 (în engleză se utilizează punctul în loc de virgula zecimală – n.t.). Din fericire, **int** este numai unul din mulţimea tipurilor de date predefinite în C#. Pentru reprezentarea numerelor fracţionare, în C# sunt definite două tipuri în virgulă mobilă: **float** şi **double**, corespunzând valorilor în simplă, respectiv dublă precizie. Dintre cele două, **double** este probabil cel mai frecvent întrebuinţat.

Pentru a declara o variabilă de tip **double**, utilizaţi o instrucţiune similară cu cea de mai jos:

```
double result;
```

Aici, **result** este numele variabilei, care este de tip **double**. Cum **result** este de un tip în virgulă mobilă, poate memora valori cum sunt 122.23, 0.034 sau –19.0.

Pentru a înţelege mai bine diferenţa dintre tipurile **int** şi **double**, încercaţi programul următor:

```
/*
   Acest program demonstreaza diferentele dintre
   int si double
*/

using System;

class Example3 {
  public static void Main() {
```

```
    int ivar;   // declaratia unei variabile int
    double dvar;  // declaratia unei variabile in virgula mobila

    ivar = 100;  // atribuim lui ivar valoarea 100

    dvar = 100.0;  // atribuim lui dvar valoarea 100.0

    Console.WriteLine("Valoarea initiala a lui ivar: " + ivar);
    Console.WriteLine("Valoarea initiala a lui dvar: " + dvar);

    Console.WriteLine(); // afisarea unei linii vide

    // ambele variabile se impart la 3
    ivar = ivar / 3;
    dvar = dvar / 3.0;

    Console.WriteLine("ivar dupa impartire: " + ivar);
    Console.WriteLine("dvar dupa impartire: " + dvar);
  }
}
```

Afişează o linie vidă

Programul va afişa rezultatul prezentat mai jos:

```
Valoarea initiala a lui ivar: 100
Valoarea initiala a lui dvar: 100

ivar dupa impartire: 33
dvar dupa impartire: 33.3333333333333
```

După cum se observă, atunci când **ivar** se împarte la 3, se efectuează o împărţire întreagă, iar rezultatul este 33 – partea fracţionară se pierde. Atunci când însă **dvar** se împarte la 3, partea fracţionară este păstrată.

Programul demonstrează că, atunci când doriţi să specificaţi o valoare în virgulă mobilă, trebuie să includeţi punctul zecimal. Altfel, valoarea va fi interpretată ca fiind întreagă. De exemplu, în C# valoarea 100 este întreagă, în timp ce 100.0 este în virgulă mobilă.

Există încă un aspect nou de observat în programul de mai sus. Pentru a afişa o linie vidă, este suficient să apelaţi **WriteLine()** fără parametri.

Sfatul expertului

Întrebare: De ce în C# există tipuri de date diferite pentru întregi şi valori în virgulă mobilă? Cu alte cuvinte, de ce nu sunt toate valorile numerice de un singur tip?

Răspuns: Limbajul C# vă pune la dispoziţie tipuri de date diferite pentru ca să puteţi scrie programe eficiente. De exemplu, operaţiile aritmetice pe numere întregi decurg mai rapid decât calculele în virgulă mobilă. În consecinţă, dacă nu aveţi neapărat nevoie de valori fracţionare, nu este cazul să vă complicaţi cu întârzierea determinată de lucrul cu tipurile **float** şi **double**. În al doilea rând, un anumit tip de date poate avea nevoie de mai puţină memorie decât altul. Punând la dispoziţie tipuri diferite, limbajul C# vă permite să utilizaţi cât mai eficient resursele sistemului. În fine, unii algoritmi impun (sau, cel puţin, sunt avantajaţi de) utilizarea unui anumit tip de date. Limbajul C# vă pune la dispoziţie numărul de tipuri predefinite care vă asigură o flexibilitate maximă.

Proiectul 1-1: Conversia temperaturii din grade Fahrenheit în grade Celsius

Chiar dacă exemplele anterioare demonstrează câteva facilităţi importante ale limbajului C#, ele nu sunt foarte utile. Chiar dacă încă nu cunoaşteţi foarte multe despre C#, puteţi aplica în practică ceea ce aţi învăţat pentru a crea un program util în mod practic. În cadrul acestui proiect, vom crea un program care converteşte temperatura de pe scara Fahrenheit pe scara Celsius.

Programul declară două variabile de tip **double**. Una va memora numărul gradelor Fahrenheit, iar cealaltă va păstra numărul gradelor Celsius rezultat în urma conversiei. După cum poate vă amintiţi de la fizică, formula de conversie a gradelor Fahrenheit în grade Celsius este:

C = 5/9 * (F – 32)

unde C reprezintă grade Celsius, iar F grade Fahrenheit.

Pas cu pas

1. Creaţi un fişier C# nou denumit **FtoC.cs**. (Dacă utilizaţi mediul integrat Visual C++ şi nu linia de comandă, va trebui să adăugaţi acest fişier la un proiect C#, cum am precizat anterior în acest modul.)
2. Introduceţi în fişier următorul program:

```
/*
   Proiectul 1-1

   Acest program converteste grade Fahrenheit in grade Celsius

   Denumiti programul FtoC.cs
*/

using System;

class FtoC {
  public static void Main() {
    double f; // temperatura in grade Fahrenheit
    double c; // temperatura in grade Celsius

    f = 59.0; // incepem cu 59 de grade Fahrenheit

    c = 5.0 / 9.0 * (f - 32.0); // conversie la grade Celsius

    Console.Write(f + " grade Fahrenheit = ");
    Console.WriteLine(c + " grade Celsius.");
  }
}
```

3. Compilați programul utilizând mediul integrat Visual C++ (urmărind instrucțiunile descrise anterior în acest modul) sau cu ajutorul următoarei linii de comandă:

```
C>csc FtoC.cs
```

4. Rulați programul din mediul Visual C++, sau utilizând comanda:

```
C>FtoC
```

Va apărea pe ecran rezultatul:

```
59 grade Fahrenheit = 15 grade Celsius
```

5. În momentul de față, programul convertește temperatura de 59 de grade Fahrenheit în grade Celsius. Modificând însă valoarea atribuită lui **f**, puteți face programul să convertească o temperatură diferită.

Exerciții la minut

- Ce cuvânt cheie există în C# pentru tipul întreg de date?
- Ce este **double**?
- Este **using System** o parte necesară într-un program C#?

Două instrucțiuni de control

În corpul unei metode, execuția se desfășoară de la o instrucțiune la alta, de la prima până la ultima. Este însă posibil să modificăm acest flux secvențial, utilizând diversele instrucțiuni de control de care dispunem în C#. Chiar dacă vom studia în amănunt instrucțiunile de control ceva mai târziu, două dintre acestea sunt prezentate pe scurt aici, deoarece le vom utiliza pentru a scrie programe de test.

Instrucțiunea if

Puteți executa selectiv părți dintr-un program utilizând instrucțiunea condițională din C#: instrucțiunea **if**. Aceasta funcționează și în C# similar cu instrucțiunea IF din orice alt limbaj. De exemplu, **if**-ul din C# este sintactic identic cu instrucțiunile **if** din C, C++ sau Java. Forma cea mai simplă a instrucțiunii este prezentată mai jos:

if(*condiție*) *instrucțiune;*

Aici, *condiție* este o expresie de tip boolean (adică poate să fie adevărată sau falsă). Atunci când *condiție* este adevărată, instrucțiunea este executată. Când însă *condiție* este falsă, fluxul execuției sare peste instrucțiune. Iată un exemplu:

```
if(10 < 11) Console.WriteLine("10 este mai mic decat 11");
```

- Cuvântul cheie în C# pentru tipul întreg de date este **int**.
- **double** este cuvântul cheie pentru tipul de date în virgulă mobilă cu precizie dublă.
- Nu, dar reprezintă un avantaj important.

În acest caz, cum 10 este mai mic decât 11, expresia condițională este adevărată, iar instrucțiunea **WriteLine()** se va executa. Să considerăm însă următoarea instrucțiune:

```
if(10 < 9) Console.WriteLine("Linie care nu este afisata");
```

În acest caz, 10 nu este mai mic decât 9. Apelul lui **WriteLine()** nu va mai avea loc.

Limbajul C# definește un set complet de operatori relaționali, care pot fi utilizați în expresiile condiționale. Iată care sunt aceștia:

Operatorul	Semnificația
<	Mai mic decât
<=	Mai mic sau egal cu
>	Mai mare decât
>=	Mai mare sau egal cu
==	Egal cu
!=	Diferit de

Remarcați că operatorul de egalitate este semnul de egalitate dublat.

Iată un program care prezintă instrucțiunea **if**:

```
// Prezentam instructiunea if

using System;

class IfDemo {
  public static void Main() {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) Console.WriteLine("a este mai mic decat b");

    // aceasta instructiune nu afiseaza nimic

    if(a == b) Console.WriteLine("Nu veti vedea nimic");

    Console.WriteLine();

    c = a - b;  // c contine -1

    Console.WriteLine("c contine -1");
    if(c >= 0) Console.WriteLine("c nu este negativ");
    if(c < 0) Console.WriteLine("c este negativ");
    Console.WriteLine();

    c = b - a;  // c contine acum 1

    Console.WriteLine("c contine 1");
    if(c >= 0) Console.WriteLine("c nu este negativ");
    if(c < 0) Console.WriteLine("c este negativ");
  }
}
```

O instructiune if

3. Compilați programul utilizând mediul integrat Visual C++ (urmărind instrucțiunile descrise anterior în acest modul) sau cu ajutorul următoarei linii de comandă:

```
C>csc FtoC.cs
```

4. Rulați programul din mediul Visual C++, sau utilizând comanda:

```
C>FtoC
```

Va apărea pe ecran rezultatul:

```
59 grade Fahrenheit = 15 grade Celsius
```

5. În momentul de față, programul convertește temperatura de 59 de grade Fahrenheit în grade Celsius. Modificând însă valoarea atribuită lui **f**, puteți face programul să convertească o temperatură diferită.

Exerciții la minut

- Ce cuvânt cheie există în C# pentru tipul întreg de date?
- Ce este **double**?
- Este **using System** o parte necesară într-un program C#?

Două instrucțiuni de control

În corpul unei metode, execuția se desfășoară de la o instrucțiune la alta, de la prima până la ultima. Este însă posibil să modificăm acest flux secvențial, utilizând diversele instrucțiuni de control de care dispunem în C#. Chiar dacă vom studia în amănunt instrucțiunile de control ceva mai târziu, două dintre acestea sunt prezentate pe scurt aici, deoarece le vom utiliza pentru a scrie programe de test.

Instrucțiunea `if`

Puteți executa selectiv părți dintr-un program utilizând instrucțiunea condițională din C#: instrucțiunea **if**. Aceasta funcționează și în C# similar cu instrucțiunea IF din orice alt limbaj. De exemplu, **if**-ul din C# este sintactic identic cu instrucțiunile **if** din C, C++ sau Java. Forma cea mai simplă a instrucțiunii este prezentată mai jos:

if(*condiție*) *instrucțiune;*

Aici, *condiție* este o expresie de tip boolean (adică poate să fie adevărată sau falsă). Atunci când *condiție* este adevărată, instrucțiunea este executată. Când însă *condiție* este falsă, fluxul execuției sare peste instrucțiune. Iată un exemplu:

```
if(10 < 11) Console.WriteLine("10 este mai mic decat 11");
```

- Cuvântul cheie în C# pentru tipul întreg de date este **int**.
- **double** este cuvântul cheie pentru tipul de date în virgulă mobilă cu precizie dublă.
- Nu, dar reprezintă un avantaj important.

În acest caz, cum 10 este mai mic decât 11, expresia condițională este adevărată, iar instrucțiunea **WriteLine()** se va executa. Să considerăm însă următoarea instrucțiune:

```
if(10 < 9) Console.WriteLine("Linie care nu este afisata");
```

În acest caz, 10 nu este mai mic decât 9. Apelul lui **WriteLine()** nu va mai avea loc.

Limbajul C# definește un set complet de operatori relaționali, care pot fi utilizați în expresiile condiționale. Iată care sunt aceștia:

Operatorul	Semnificația
<	Mai mic decât
<=	Mai mic sau egal cu
>	Mai mare decât
>=	Mai mare sau egal cu
==	Egal cu
!=	Diferit de

Remarcați că operatorul de egalitate este semnul de egalitate dublat.

Iată un program care prezintă instrucțiunea **if**:

```
// Prezentam instructiunea if

using System;

class IfDemo {
  public static void Main() {
    int a, b, c;

    a = 2;
    b = 3;

    if(a < b) Console.WriteLine("a este mai mic decat b");

    // aceasta instructiune nu afiseaza nimic

    if(a == b) Console.WriteLine("Nu veti vedea nimic");

    Console.WriteLine();

    c = a - b;    // c contine -1

    Console.WriteLine("c contine -1");
    if(c >= 0) Console.WriteLine("c nu este negativ");
    if(c < 0) Console.WriteLine("c este negativ");

    Console.WriteLine();

    c = b - a;    // c contine acum 1

    Console.WriteLine("c contine 1");
    if(c >= 0) Console.WriteLine("c nu este negativ");
    if(c < 0) Console.WriteLine("c este negativ");
  }
}
```

O instructiune **if**

Rezultatul afişat de acest program este prezentat mai jos:

```
a este mai mic decat b

c contine -1
c este negativ

c contine 1
c nu este negativ
```

Să observăm şi un alt amănunt în acest program. Linia

```
int a, b, c;
```

declară trei variabile **a**, **b** şi **c**, utilizând o listă cu elementele separate prin virgule. După cum am precizat anterior, atunci când aveţi nevoie de două sau mai multe variabile de acelaşi tip, acestea pot fi declarate într-o singură instrucţiune. Este suficient să separaţi numele variabilelor prin virgule.

Bucla for

Este posibil să executaţi în mod repetat o secvenţă de cod prin crearea unei *bucle*. Limbajul C# dispune de o varietate de construcţii repetitive puternice. Cea despre care vom discuta aici este bucla **for**. Dacă ştiţi unul din limbajele C, C++ sau Java, veţi fi încântat să aflaţi că bucla **for** din C# funcţionează la fel ca şi în aceste limbaje. Cea mai simplă formă a buclei **for** este prezentată mai jos:

for(*iniţializare; condiţie; iteraţie) instrucţiune;*

În forma cea mai întâlnită, partea de *iniţializare* atribuie unei variabile de control a buclei o valoare iniţială. *Condiţia* este o expresie de tip boolean care testează variabila de control a buclei. Dacă rezultatul testului este adevărat, atunci bucla **for** continuă să itereze. Când acesta este însă fals, bucla se termină. Expresia *iteraţie* determină maniera în care variabila de control a buclei se modifică la fiecare iteraţie. Iată un program scurt care utilizează bucla **for**:

```
// Prezentam bucla for

using System;

class ForDemo {
  public static void Main() {
    int count;

    for(count = 0; count < 5; count = count + 1)
      Console.WriteLine("Valoarea lui count: " + count);

    Console.WriteLine("Gata!");
  }
}
```

Această buclă iterează de 5 ori.

Rezultatul generat pe ecran de program este prezentat mai jos:

```
Valoarea lui count: 0
Valoarea lui count: 1
Valoarea lui count: 2
Valoarea lui count: 3
Valoarea lui count: 4
Gata!
```

În acest exemplu, variabila de control a buclei este **count**. Aceasta primeşte valoarea zero în secţiunea de iniţializare a lui **for**. La începutul fiecărei iteraţii (inclusiv la prima), se efectuează testul condiţiei **count < 5**. Dacă rezultatul acestui test este adevărat, atunci se execută instrucţiunea **WriteLine()** şi apoi secţiunea de iteraţie a buclei. Acest proces continuă până când testul condiţional devine fals, moment în care execuţia se reia cu ceea ce se găseşte după buclă.

Ca un amănunt interesant, în programele C# scrise în mod profesionist nu veţi întâlni aproape niciodată secţiunea de iteraţie a buclei scrisă ca în programul anterior. Cu alte cuvinte, foarte rar veţi întâlni instrucţiuni de forma:

```
count = count + 1;
```

Aceasta se întâmplă fiindcă în C# dispunem de un operator special de incrementare, care efectuează această operaţie mai eficient. Operatorul de incrementare este ++ (adică semnul + dublat). Efectul operatorului de incrementare este de a creşte valoarea operandului cu o unitate. Utilizând operatorul de incrementare, instrucţiunea precedentă se poate rescrie astfel:

```
count++;
```

În consecinţă, bucla **for** din programul anterior se va scrie de regulă astfel:

```
for(count = 0; count < 5; count++)
```

Efectuaţi această modificare. După cum veţi observa, bucla va rula exact la fel ca înainte.

În C#, dispunem şi de un operator de decrementare, notat - - (semnul minus dublat). Acest operator scade o unitate din valoarea operandului său.

Exerciţii la minut

- Ce face instrucţiunea **if**?
- Ce face instrucţiunea **for**?
- Care sunt operatorii relaţionali în C#?

- **if** este instrucţiunea condiţională.
- **for** este una din instrucţiunile repetitive din C#.
- Operatorii relaţionali sunt ==, !=, <, >, <= şi >=.

Utilizarea blocurilor de instrucțiuni

Un alt element esențial în C# este *blocul de cod*. Un bloc de cod este o grupare de două sau mai multe instrucțiuni. Aceasta se realizează prin includerea instrucțiunilor între o acoladă deschisă și una închisă. După creare, un bloc de cod devine o unitate logică și poate fi utilizat oriunde în locul unei singure instrucțiuni. Spre exemplu, un bloc poate constitui ținta unor instrucțiuni **if** și **for**. Să considerăm următoarea instrucțiune **if**:

```
if(w < h) {
  v = w * h;
  w = 0;
}
```

Aici, dacă **w** are valoarea mai mică decât **h**, atunci se execută ambele instrucțiuni din bloc. În concluzie, cele două instrucțiuni din interiorul blocului formează o unitate logică, nici una dintre instrucțiuni neputându-se executa separat de cealaltă. Aspectul esențial aici este că ori de câte ori aveți nevoie să grupați logic două sau mai multe instrucțiuni, aceasta se realizează prin crearea unui bloc. Blocurile de cod permit implementarea multor algoritmi într-o manieră mai clară și mai eficientă.

Iată în continuare un program care utilizează un bloc de cod pentru a evita o împărțire la zero:

```
// Prezinta blocurile de cod

using System;

class BlockDemo {
  public static void Main() {
    int i, j, d;

    i = 5;
    j = 10;

    // tinta acestui if este un bloc
    if(i != 0) {
      Console.WriteLine("i nu este egal cu zero");
      d = j / i;
      Console.WriteLine("j / i este " + d);
    }
  }
}
```

Ținta lui **if** este acest intreg bloc.

Sfatul expertului

Întrebare: Utilizarea blocurilor de cod conduce la ineficiență la rularea programului? Cu alte cuvinte, prezența acoladelor determină consumul unui timp suplimentar pe parcursul execuției programului?

Răspuns: Nu. Blocurile de cod nu adaugă în nici un caz vreo întârziere. De fapt, datorită capacității lor de a simplifica transpunerea în cod a anumitor algoritmi, utilizarea lor determină în general o creștere a vitezei și eficienței.

Rezultatul generat de acest program este prezentat mai jos:

```
i nu este egal cu zero
j / i este 2
```

În acest caz, ținta instrucțiunii **if** este un bloc de cod și nu doar o singură instrucțiune. Atunci când condiția care controlează instrucțiunea **if** este adevărată (iar în acest caz este), se execută cele trei instrucțiuni din bloc. Încercați să dați lui **i** valoarea zero și observați rezultatul.

După cum veți observa mai târziu în această carte, blocurile de cod au și alte proprietăți și întrebuințări. Principalul motiv al existenței lor este însă crearea unităților atomice (adică inseparabile logic) de cod.

Separatori și poziționare

În C#, punctul și virgula marchează sfârșitul unei instrucțiuni. Cu alte cuvinte, fiecare instrucțiune separată trebuie să se termine cu punct și virgulă.

După cum știți, un bloc reprezintă un set de instrucțiuni legate logic, incluse între o acoladă deschisă și una închisă. Blocurile *nu* se termină cu punct și virgulă. Deoarece un bloc este un grup de instrucțiuni, cu punct și virgulă după fiecare instrucțiune, este normal ca un bloc să nu se termine cu punct și virgulă; finalul blocului este indicat de acolada închisă.

În C#, sfârșitul de linie nu este recunoscut ca sfârșit de instrucțiune – doar punctul și virgula pot marca terminarea unei instrucțiuni. Din acest motiv, nu are nici o importanță unde poziționați o instrucțiune în cadrul unei linii. De exemplu, secvența din C#:

```
x = y;
y = y + 1;
Console.WriteLine(x + " " + y);
```

este aceeași cu:

```
x = y; y = y + 1; Console.WriteLine(x + " " + y);
```

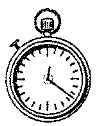
Mai mult, elementele individuale ale unei instrucțiuni pot fi poziționate pe linii separate. Spre exemplu, următoarea instrucțiune este perfect corectă:

```
Console.WriteLine("Aceasta este o linie lunga de text" +
                   x + y + z +
                  "inca niste text");
```

Despărțirea liniilor lungi în această manieră este deseori utilizată pentru a face programele mai ușor de citit. În acest mod se poate evita și despărțirea automată a liniilor excesiv de lungi.

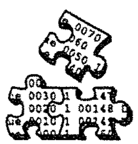
Indentarea programelor

Poate că ați observat în exemplele anterioare că anumite instrucțiuni erau scrise indentat. Limbajul C# este independent de format, în sensul că nu are importanță unde poziționați instrucțiunile unele în raport cu celelalte pe o linie. De-a lungul anilor însă, s-a dezvoltat un stil de indentare comun și universal acceptat, care permite scrierea unor programe foarte ușor de citit. Această lucrare adoptă acest stil și se recomandă ca și dumneavoastră să procedați la fel. În conformitate cu acest stil, se trece la un nivel nou de indentare după fiecare acoladă deschisă, revenind cu un nivel înapoi după fiecare acoladă închisă. Există anumite instrucțiuni care favorizează o indentare suplimentară; acestea vor fi discutate ulterior.



Exerciții la minut

- Cum se creează un bloc de cod? Care este rolul acestuia?
- În C#, instrucțiunile se termină cu ____________.
- Toate instrucțiunile C# trebuie să înceapă și să se termine pe o singură linie. Este adevărat sau fals?



Proiectul 1-2: Îmbunătățirea programului de conversie a temperaturii

Puteți acum utiliza bucla **for**, instrucțiunea **if** și blocurile de cod pentru a crea o versiune îmbunătățită a convertorului din grade Fahrenheit în grade Celsius pe care l-ați dezvoltat în primul proiect. Această nouă versiune va afișa o tabelă de conversie, începând cu 0 grade Fahrenheit și până la 99. La fiecare 10 grade, se va afișa o linie vidă. Aceasta se realizează utilizând o variabilă numită **counter**, care numără liniile afișate. Fiți deosebit de atent la utilizarea acesteia.

- Un bloc începe cu o acoladă deschisă {. Se termină cu o acoladă închisă }. Un bloc creează o unitate atomică de cod.
- Instrucțiunile C# se termină cu punct și virgulă.
- Fals.

Pas cu pas

1. Creați un nou fișier, denumit **FtoCTable.cs**.
2. Introduceți următorul program în fișier:

```
/*
   Proiectul 1-2

   Acest program afiseaza o tabela de conversie
   din grade Fahrenheit in grade Celsius

   Denumiti programul FtoCTable.cs.
*/

using System;

class FtoCTable {
  public static void Main() {
    double f, c;
    int counter;

    counter = 0;
    for(f = 0.0; f < 100.0; f++) {
      c = 5.0/9.0 * (f - 32.0);   // conversie in grade Celsius
      Console.WriteLine(f + " grade Fahrenheit = " +
                        c + " grade Celsius.");

      counter++;   ◄── Incrementăm contorul la fiecare iterație.

      // la fiecare 10 linii, afisam una vida

      if(counter == 10) {   ◄── Când counter este 10, afișăm o linie vidă.
        Console.WriteLine();

        counter = 0;   // resetare a contorului de linii
      }
    }
  }
}
```

3. Compilați programul utilizând mediul integrat Visual C++ sau următoarea linie de comandă:

```
C>csc FtoCTable.cs
```

4. Rulați programul utilizând mediul integrat Visual C++ sau următoarea linie de comandă:

```
C>FtocTable
```

Iată o parte din rezultatele pe care le veți vedea afișate pe ecran. Observați că rezultatele care nu produc valori exacte conțin o parte fracționară.

```
0 grade Fahrenheit = -17.7777777777778 grade Celsius
1 grade Fahrenheit = -17.2222222222222 grade Celsius
2 grade Fahrenheit = -16.6666666666667 grade Celsius
3 grade Fahrenheit = -16.1111111111111 grade Celsius
4 grade Fahrenheit = -15.5555555555556 grade Celsius
5 grade Fahrenheit = -15 grade Celsius.
6 grade Fahrenheit = -14.4444444444444 grade Celsius
7 grade Fahrenheit = -13.8888888888889 grade Celsius
```

```
8 grade Fahrenheit = -13.3333333333333 grade Celsius
9 grade Fahrenheit = -12.7777777777778 grade Celsius

10 grade Fahrenheit = -12.2222222222222 grade Celsius
11 grade Fahrenheit = -11.6666666666667 grade Celsius
12 grade Fahrenheit = -11.1111111111111 grade Celsius
13 grade Fahrenheit = -10.5555555555556 grade Celsius
14 grade Fahrenheit = -10 grade Celsius
15 grade Fahrenheit = -9.4444444444444 grade Celsius
16 grade Fahrenheit = -8.8888888888889 grade Celsius
17 grade Fahrenheit = -8.3333333333333 grade Celsius
18 grade Fahrenheit = -7.7777777777778 grade Celsius
19 grade Fahrenheit = -7.2222222222222 grade Celsius
```

Cuvintele cheie din C#

Există 77 de cuvinte cheie definite în limbajul C# (vezi tabelul 1-1). Aceste cuvinte cheie, împreună cu sintaxa operatorilor şi separatorii, alcătuiesc definiţia limbajului C#. Aceste cuvinte nu pot fi utilizate ca nume de variabile, clase sau metode.

Tabelul 1-1 Cuvintele cheie din C#

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	volatile
void	while			

Identificatori

În C#, un identificator este un nume atribuit unei metode, unei variabile sau oricărei alte entităţi definite de utilizator. Identificatorii pot avea unul sau mai multe caractere. Numele variabilelor pot începe cu orice literă din alfabet sau cu underscore (_). Acest simbol este utilizat pentru a face numele unei variabile mai uşor de citit, ca de exemplu **line_count**. Literele mari diferă de cele mici; cu alte cuvinte, în C#, **myvar** şi **MyVar** reprezintă variabile diferite. Iată câteva exemple de identificatori corect formaţi:

```
Test    x       y2       MaxLoad
up      _top    my_var   sample23
```

Reţineţi că un identificator nu poate începe cu o cifră. În concluzie, **12x** este un identificator incorect. Regulile nescrise ale unei programări corecte cer să utilizaţi nume de identificatori care reflectă sensul sau modul de utilizare al entităţilor denumite.

Chiar dacă în C# cuvintele cheie nu pot fi utilizate ca nume de identificatori, limbajul vă permite să prefixaţi aceste cuvinte cu simbolul @, obţinând identificatori corecţi. Spre exemplu, **@for** este un identificator valabil. În această situaţie, identificatorul se numeşte de fapt **for**, iar simbolul @ este ignorat. De regulă, utilizarea cuvintelor cheie precedate de @ pe post de identificatori nu este recomandată, exceptând anumite scopuri speciale. Din punct de vedere tehnic, simbolul @ poate precede orice identificator, dar această practică este considerată un abuz.

Exerciţii la minut

- Care dintre următoarele este un cuvânt cheie: **for**, **For** sau **FOR**?
- Ce tipuri de caractere pot apărea într-un identificator C#?
- Reprezintă **index21** şi **Index21** acelaşi identificator?

Biblioteca de clase C#

Programele prezentate ca exemple în acest capitol utilizează două metode predefinite în C#: **WriteLine()** şi **Write()**. După cum am stabilit, aceste metode sunt membri ai clasei **Console**, care face parte din spaţiul de nume **System**, definit de biblioteca de clase a arhitecturii .NET. După cum s-a explicat anterior în acest modul, mediul C# se bazează pe biblioteca de clase a arhitecturii .NET pentru a asigura unele facilităţi cum ar fi operaţiile de intrare-ieşire, lucrul cu şiruri de caractere, lucrul în reţea sau dezvoltarea de interfeţe grafice. În concluzie, C# văzut ca un întreg reprezintă suma dintre limbajul C# propriu-zis şi clasele standard .NET. După cum veţi vedea, biblioteca de clase asigură o mare parte din funcţionalitatea oricărui program C#. Într-adevăr, a deveni programator în C# presupune în bună parte deprinderea lucrului cu aceste clase standard. Pe parcursul cărţii, sunt prezentate diferite elemente ale claselor şi metodelor din biblioteca .NET. Biblioteca .NET este însă de mari dimensiuni, fiind mai adecvată studiului individual.

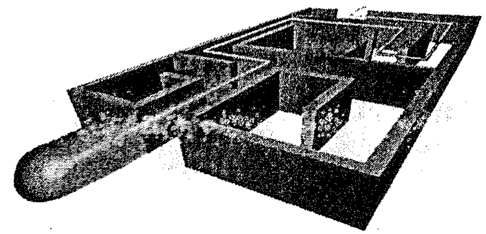
- Cuvântul cheie este **for**. Toate cuvintele cheie din C# se scriu cu litere mici.
- Un identificator C# poate conţine litere, cifre şi underscore.
- Nu, limbajul C# este sensibil la tipurile de litere.

Verificarea cunoştinţelor

1. Ce este MSIL şi ce importanţă prezintă pentru C#?
2. Ce este motorul comun de programare?
3. Care sunt cele trei principii fundamentale ale programării orientate spre obiecte?
4. Cu ce începe execuţia programelor C#?
5. Ce este o variabilă? Ce este un spaţiu de nume?
6. Care dintre următoarele nume de variabile sunt incorecte?
   A. `count`
   B. `$count`
   C. `count27`
   D. `67count`
   E. `@if`
7. Cum puteţi crea un comentariu pe o singură linie? Cum puteţi crea un comentariu multilinie?
8. Prezentaţi forma generală a instrucţiunii `if`. Prezentaţi forma generală a buclei `for`.
9. Cum puteţi crea un bloc de cod?
10. Este obligatoriu ca fiecare program C# să înceapă cu următoarea instrucţiune?
    `using System;`
11. Acceleraţia gravitaţională pe Lună este aproximativ egală cu 17 procente din cea de pe Pământ. Scrieţi un program care calculează greutatea dumneavoastră pe Lună.
12. Modificaţi proiectul 1-2 astfel încât să afişeze o tabelă de conversie din ţoli (inches) în metri. Afişaţi conversia pe un interval de 4 metri, din ţol în ţol. Afişaţi câte o linie vidă la fiecare 12 ţoli. (Un metru este egal cu aproximativ 39.37 ţoli.)

Modulul 2

Prezentarea tipurilor de date şi operatorilor



Scopurile acestui modul

- **Însuşirea tipurilor fundamentale din C#**
- **Utilizarea literalilor**
- **Crearea variabilelor iniţializate**
- **Cunoaşterea domeniului de valabilitate al unei metode**
- **Înţelegerea conversiei de tipuri şi castului**
- **Învăţarea operatorilor aritmetici**
- **Învăţarea operatorilor relaţionali şi logici**
- **Studiul operatorului de atribuire**
- **Studiul expresiilor**

Tipurile de date şi operatorii stau la baza oricărui limbaj de programare şi nici C# nu face excepţie de la această regulă. Aceste elemente stabilesc limitele unui limbaj şi determină tipurile de activităţi pentru care poate fi utilizat. După cum vă aşteptaţi, limbajul C# oferă o gamă largă atât de tipuri de date, cât şi de operatori, fiind astfel adecvat pentru dezvoltarea unor categorii diverse de programe.

Tipurile de date şi operatorii reprezintă un subiect vast. Vom începe prin a examina tipurile de date fundamentale din C# şi operatorii cel mai des utilizaţi. Vom arunca de asemenea o privire amănunţită asupra variabilelor şi vom studia expresiile.

De ce sunt tipurile de date importante

Tipurile de date au în C# o importanţă deosebită, deoarece acesta este un limbaj puternic tipizat. Aceasta înseamnă că pentru toate operaţiile, compilatorul efectuează verificări privind compatibilitatea tipurilor. Operaţiile nepermise nu sunt compilate. Verificarea puternică a tipurilor ajută deci la prevenirea erorilor şi creşte fiabilitatea programelor. Pentru a permite verificarea puternică a tipurilor, tuturor variabilelor, expresiilor şi valorilor li se asociază un tip. Nu există conceptul de variabilă „fără tip". Mai mult, tipul unei valori determină în mod exact operaţiile permise asupra sa. Unele operaţii permise asupra anumitor tipuri nu sunt permise şi asupra altora.

Tipuri valorice în C#

Limbajul C# include două categorii generale de tipuri predefinite: *tipuri valorice* și *tipuri referință*. Tipurile referință din C# sunt definite de clase; discuția despre clase o amânăm pentru mai târziu. La baza limbajului C# stau însă cele 13 tipuri valorice, prezentate în tabelul 2-1. Tipurile valorice sunt cunoscute și sub denumirea de *tipuri simple*.

Tabelul 2-1 Tipurile valorice din C#

Tipul	Semnificația
`bool`	Reprezintă valorile de adevăr (adevărat/fals)
`byte`	Întregi pe 8 biți, fără semn
`char`	Caractere
`decimal`	Tip numeric pentru calcule financiare
`double`	Virgulă mobilă, dublă precizie
`float`	Virgulă mobilă, simplă precizie
`int`	Numere întregi
`long`	Întregi în formă lungă
`sbyte`	Întregi pe 8 biți, cu semn
`short`	Întregi în formă scurtă
`uint`	Întregi fără semn
`ulong`	Întregi în formă lungă, fără semn
`ushort`	Întregi în formă scurtă, fără semn

Limbajul C# precizează în mod strict un domeniu de valori și un comportament pentru fiecare dintre tipurile valorice. Din rațiuni de portabilitate, în C# nu s-au admis compromisuri la acest aspect. De exemplu, un **int** rămâne la fel, indiferent de mediul de execuție. Nu este necesar, deci, să rescriem codul pentru adaptarea la o anumită platformă. Chiar dacă precizarea strictă a dimensiunii tipurilor valorice conduce la o ușoară degradare a performanțelor în anumite medii, aceasta este necesară pentru asigurarea portabilității.

Întregi

În C# sunt definite nouă tipuri întregi: **char, byte, sbyte, short, ushort, int, uint, long** și **ulong**. Tipul **char** este însă utilizat în primul rând la reprezentarea caracterelor și de aceea va fi prezentat ulterior în acest modul. Celelalte opt tipuri rămase sunt utilizate pentru calcule numerice. Dimensiunile lor în biți și domeniile de valori sunt prezentate în continuare:

Tipul	Lărgimea în biți	Domeniul
`byte`	8	0 la 255
`sbyte`	8	-128 la 127
`short`	16	-32 768 la 32 767
`ushort`	16	0 la 65 535
`int`	32	-2 147 483 648 la 2 147 483 647
`uint`	32	0 la 4 294 967 295
`long`	64	- 9 223 372 036 854 775 808 la 9 223 372 036 854 775 807
`ulong`	64	0 la 18 446 744 073 709 551 615

După cum se observă în tabel, în C# sunt definite atât versiuni cu semn, cât și fără semn pentru diferitele tipuri întregi. Diferența între întregii cu semn și cei fără semn apare la interpretarea bitului cel mai semnificativ. Dacă specificăm un întreg cu semn, compilatorul de C# va genera cod care presupune că bitul cel mai semnificativ al întregului este folosit ca *bit de semn*. Dacă bitul de semn este 0, atunci numărul este pozitiv; dacă este 1, atunci numărul este negativ. Numerele negative sunt aproape întotdeauna reprezentate utilizând metoda *complementului față de doi*. În această reprezentare, se inversează toți biții numărului, cu excepția celui de semn, adunându-se 1 la numărul rezultat. În fine, bitul de semn primește valoarea 1.

Numerele întregi cu semn sunt importante pentru o mare varietate de algoritmi, dar ele pot atinge numai pe jumătate valorile maxime absolute atinse de echivalentele lor fără semn. Spre exemplu, iată reprezentarea ca **short** a numărului 32 767:

0 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1

În cazul unei valori cu semn, dacă dăm bitului de semn valoarea 1, numărul reprezentat este interpretat ca –1 (în formatul complementului față de doi). Dacă însă declarăm această valoare ca **ushort**, când bitul cel mai semnificativ devine 1, numărul este interpretat ca 65 535.

Cel mai des utilizat tip întreg este probabil tipul **int**. Variabilele de tip **int** se utilizează de regulă pentru controlul buclelor, pentru indexarea tablourilor, dar și pentru aritmetica obișnuită cu numere întregi. Atunci când aveți nevoie de un întreg cu un domeniu mai larg decât **int**, dispuneți de multe opțiuni. Dacă valoarea pe care trebuie să o memorați este fără semn, puteți utiliza **uint**. Pentru valori mari cu semn, alegeți **long**. În cazul valorilor mari fără semn, utilizați **ulong**.

Iată în continuare un program care calculează numărul de cuburi cu latura de un țol care încap într-un cub cu latura de o milă. Din cauză că valoarea este foarte mare, programul utilizează o variabilă de tip **long** pentru a o memora.

```
/*
   Calculul numarului de cuburi cu latura de 1 tol
   care intra intr-un cub cu latura de 1 mila
*/

using System;
```

```
class Inches {
  static void Main() {
    long ci;
    long im;

    im = 5280 * 12;
    ci = im * im * im;

    Console.WriteLine("Intr-un cub cu latura de 1 mila incap "
                      + ci + " cuburi cu latura de 1 tol.");
  }
}
```

Notă:
1 milă = 5280 picioare
1 picior = 12 ţoli

Iată ce afişează acest program:

```
Intr-un cub cu latura de 1 mila incap 254358061056000 cuburi cu latura de 1 tol.
```

În mod evident, acest rezultat nu putea fi memorat într-o variabilă de tip **int** sau **uint**.

Cele mai mici tipuri întregi sunt **byte** şi **sbyte**. Tipul **byte** poate conţine valori fără semn cuprinse între 0 şi 255. Variabilele de tip **byte** sunt utile mai ales atunci când se lucrează cu date binare brute, cum ar fi un flux de date produs de un anumit dispozitiv fizic. Pentru a reprezenta întregi mici cu semn, puteţi alege tipul **sbyte**. Iată un exemplu care utilizează o variabilă de tip **byte** la controlul unei bucle **for** care calculează suma primelor 100 de numere naturale:

```
// Utilizarea tipului byte

using System;

class Use_byte {
  static void Main() {
    byte x;
    int sum;

    sum = 0;
    for(x = 1; x <= 100; x++)
      sum = sum + x;

    Console.WriteLine("Suma primelor 100 de numere naturale este " + sum);
  }
}
```

Rezultatul afişat de program este prezentat mai jos:

```
Suma primelor 100 de numere naturale este 5050
```

Deoarece bucla for iterează numai între 0 şi 100, domeniu cuprins în domeniul de valori al tipului **byte**, nu este cazul să întrebuinţăm o variabilă de un tip mai mare.

Atunci când aveţi nevoie de un întreg mai mare decât **byte** sau **sbyte**, dar mai mic decât **int** sau **uint**, utilizaţi **short**, respectiv **ushort**.

Tipuri în virgulă mobilă

După cum am explicat în modulul 1, tipurile în virgulă mobilă se utilizează pentru a reprezenta numere care au parte fracţionară. Există două tipuri în virgulă mobilă, **float** şi **double**, care reprezintă numere fracţionare în simplă, respectiv dublă precizie. Tipul **float** are lungimea de 32 de biţi şi domeniul între 1.5E-45 şi 3.4E+38. Tipul **double** are 64 de biţi, domeniul fiind cuprins între 5E-324 şi 1.7E+308.

Dintre cele două tipuri, **double** este cel mai frecvent întrebuinţat. Unul dintre motive este acela că multe dintre funcţiile matematice din biblioteca de clase C# (care este de fapt biblioteca arhitecturii .NET) utilizează valori de tip **double**. Spre exemplu, metoda **Sqrt()** (definită în clasa standard **System.math**) întoarce o valoare de tip **double** care este rădăcina pătrată a parametrului său. În exemplul următor, vom utiliza **Sqrt()** pentru a calcula lungimea ipotenuzei unui triunghi dreptunghic cunoscând lungimile catetelor:

```
/*
   Utilizarea teoremei lui Pitagora la determinarea
   lungimii ipotenuzei, cunoscand lungimile catetelor.
*/

using System;

class Hypot {
  static void Main() {
    double x, y, z;

    x = 3;
    y = 4;

    z = Math.Sqrt(x*x + y*y);

    Console.WriteLine("Ipotenuza este " + z);
  }
}
```

Observaţi cum este apelată **Sqrt()**: precedată de numele clasei în care este membru.

Rezultatul acestui program este prezentat în continuare:

```
Ipotenuza este 5
```

Merită remarcat şi un alt aspect în exemplul anterior. După cum am amintit, metoda **Sqrt()** este un membru al clasei **Math**. Reţineţi modul în care **Sqrt()** a fost apelată, fiind precedată de numele **Math**. În mod similar, **Console** precede **WriteLine()**. Deşi nu toate metodele standard sunt apelate specificând numele clasei înainte, unele dintre ele sunt.

Tipul `decimal`

Cel mai interesant tip numeric din C# este foarte probabil tipul **decimal**, destinat utilizării la calcule monetare. Tipul decimal utilizează 128 de biţi pentru reprezentarea valorilor cuprinse între 1E-28 şi 7.9E+28. După cum probabil ştiţi,

aritmetica obişnuită în virgulă mobilă conduce la o varietate de erori de rotunjire atunci când lucrează cu valori fracţionare. Tipul **decimal** elimină aceste erori, putând reprezenta în mod precis până la 28 de poziţii zecimale (chiar 29 în unele cazuri). Această capacitate de reprezentare a valorilor zecimale fără erori de rotunjire îşi găseşte cu precădere utilitatea la calcule care implică bani.

Iată un program care utilizează tipul **decimal** într-un calcul financiar. Programul calculează un sold după aplicarea dobânzii.

```
/*
   Utilizarea tipului decimal intr-un calcul financiar (in dolari)
*/
using System;

class UseDecimal {
  public static void Main() {
    decimal balance;
    decimal rate;

    // calculul noului sold
    balance = 1000.10m;
    rate = 0.1m;
    balance = balance * rate + balance;

    Console.WriteLine("Noul sold: " + balance + " USD");
  }
}
```

toate valorile de tip **decimal** trebuie urmate de m sau M.

Sfatul expertului

Întrebare: Celelalte limbaje de programare în care am lucrat nu au un tip de date decimal. Este acest tip de date prezent numai în C# ?

Răspuns: Tipul **decimal** nu există în C, C++ sau Java. Prin urmare, din familia sa, C# este singurul limbaj care oferă acest tip.

Programul va afişa următorul rezultat:

```
Noul sold: 1100.11 USD
```

În acest program, remarcaţi că toate constantele de tipul **decimal** sunt urmate de sufixul **m** sau **M**. Aceasta este necesar fiindcă fără sufix, valorile ar fi interpretate ca fiind constante standard în virgulă mobilă, care nu sunt compatibile cu tipul de date **decimal**. (Vom examina mai amănunţit modul de specificare a constantelor numerice ulterior în acest modul.)

Exerciţii la minut

- Care sunt tipurile întregi din C# ?
- Care sunt cele două tipuri în virgulă mobilă ?
- De ce este tipul **decimal** important pentru calculele financiare ?

Caractere

În C#, caracterele nu sunt reprezentate pe 8 biţi ca în alte limbaje de programare, ca de exemplu în C++. În C# se utilizează modelul Unicode. Modelul *Unicode* defineşte un set de caractere care poate reprezenta caracterele din toate limbile de pe Pământ. Tipul **char** este deci în C# un tip fără semn cu lungimea de 16 biţi, acoperind domeniul dintre 0 şi 65 535. Setul standard de caractere ASCII pe 8 biţi este un subset al modelului Unicode, fiind cuprins între 0 şi 127. Prin urmare, caracterele ASCII continuă să fie valabile şi în C#.

Putem atribui o valoare unei variabile de tip caracter dacă includem caracterul între apostrofuri simple. De exemplu, secvenţa următoare atribuie 'X' variabilei **ch**:

```
char ch;
ch = 'X';
```

Puteţi afişa un caracter utilizând o instrucţiune **WriteLine()**. Spre exemplu, linia următoare afişează valoarea lui **ch**:

```
Console.WriteLine("Valoarea lui ch: " + ch);
```

Deşi **char** este definit în C# ca un tip întreg, nu poate fi amestecat la întâmplare cu valori întregi în toate cazurile. Aceasta se întâmplă deoarece nu se efectuează conversii automate între tipurile întregi şi **char**. De exemplu, secvenţa următoare este incorectă:

```
char ch;
ch = 10;    // eroare, nu merge
```

Motivul pentru care instrucţiunea anterioară nu funcţionează este că 10 reprezintă o valoare întreagă şi nu este convertită automat la tipul **char**. Atribuirea de mai sus conţine deci tipuri incompatibile. Dacă veţi încerca să compilaţi acest cod, se va afişa un mesaj de eroare.

Pe parcursul acestui modul, veţi vedea o modalitate de a depăşi această restricţie.

- Tipurile întregi din C# sunt **byte**, **short**, **int**, **long**, **sbyte**, **ushort**, **uint** şi **ulong**. Tehnic vorbind, şi tipul **char** este tot un tip numeric, dar este utilizat în general pentru memorarea caracterelor.
- Tipurile în virgulă mobilă sunt **float** şi **double**.
- Tipul **decimal** este extrem de preţios la calcule financiare deoarece este imun la erorile de rotunjire care afectează tipurile **float** şi **double**.

Sfatul expertului

Întrebare: De ce se utilizează modelul Unicode în C# ?

Răspuns: Limbajul C# a fost conceput pentru a permite programelor să fie scrise astfel încât să fie utilizate oriunde în lume. Limbajul trebuie deci să utilizeze un set de caractere care poate reprezenta semnele din toate limbile lumii. Unicode este setul standard de caractere conceput special cu acest scop. Desigur că utilizarea setului Unicode este ineficientă pentru limbi cum sunt engleza, germana, spaniola sau franceza, care au caractere ce „încap" în 8 biți. Acesta este însă prețul plătit pentru obținerea unei portabilități globale.

Tipul `bool`

Tipul **bool** reprezintă valorile de adevăr „adevărat" (true) și „fals" (false). În C#, acestea sunt definite prin cuvintele rezervate **true** și **false**. Orice variabilă sau expresie de tipul **bool** va lua deci una din aceste valori. În plus, nu este definită nici o regulă de conversie între tipul **bool** și valori întregi. Spre exemplu, 1 nu este convertit în **true** și nici 0 în **false**.

Iată un program care prezintă tipul **bool**:

```
// Prezentarea valorilor booleene

using System;

class BoolDemo {
  static void Main() {
    bool b;

    b = false;
    Console.WriteLine("b este " + b);
    b = true;
    Console.WriteLine("b este " + b);

    // o valoare booleana poate controla o instructiune if
    if(b) Console.WriteLine("Instructiune executata.");

    b = false;
    if(b) Console.WriteLine("Instructiune care nu se executa.");

    // rezultatul unui operator relational este o valoare booleana
    Console.WriteLine("10 > 9 este " + (10 > 9));
  }
}
```

O singură valoare **bool** poate controla o instrucțiune **if**.

Programul va afișa următorul rezultat:

```
b este false
b este true
Instructiune executata.
10 > 9 este true
```


Sunt trei aspecte interesante care merită remarcate în acest program. Mai întâi, după cum observați, atunci când se afișează o valoare booleană cu **WriteLine()**, pe ecran apar șirurile de caractere „true", respectiv „false". În al doilea rând, valoarea unei variabile de tip **bool** este suficientă, fără alte elemente în plus, pentru a controla o instrucțiune **if**. Nu este necesar să scriem o instrucțiune **if** în modul următor:

```
if(b == true) ...
```

În al treilea rând, rezultatul unui operator relațional, cum este <, este o valoare de tip **bool**. Din acest motiv, expresia **10 > 9** afișează valoarea „true". În altă ordine de idei, expresia **10 > 9** trebuie inclusă într-un set suplimentar de paranteze, deoarece operatorul + are o precedență superioară lui >.

Exerciții la minut

- Ce este Unicode ?
- Ce valori poate lua o variabilă de tip **bool** ?
- Care este lungimea în biți a unui **char** ?

Câteva opțiuni de afișare

Înainte de a continua studiul tipurilor de date și operatorilor, este util să facem o scurtă paranteză. Până acum, la afișarea unor liste de elemente, am separat fiecare parte a listei prin semnul plus, ca în exemplul:

```
Console.WriteLine("Ati comandat " + 2 + " bucati a " + 5000 + " lei fiecare.");
```

Deși este foarte convenabil, acest mod de afișare a informației numerice nu ne oferă nici un control asupra formatului în care apare informația. De exemplu, în cazul unei valori în virgulă mobilă, nu putem controla numărul de zecimale afișate. Să considerăm instrucțiunea:

```
Console.WriteLine("Valoarea lui 10/3: " + 10.0/3.0);
```

Aceasta afișează rezultatul:

```
Valoarea lui 10/3: 3.33333333333333
```

Chiar dacă este satisfăcător în anumite situații, afișarea unui număr atât de mare de zecimale poate să nu fie adecvată în altele. De exemplu, în calcule financiare, trebuie de regulă să afișăm două poziții zecimale.

- Unicode este un set de caractere internațional, pe 16 biți.
- Variabilele de tipul **bool** pot fi **true** sau **false**.
- Tipul **char** are lungimea de 16 biți.

Pentru a controla formatarea datelor numerice, este necesar să utilizăm o a doua formă a lui `WriteLine()`, prezentată mai jos, care ne permite să adăugăm și informație de formatare:

```
WriteLine("sir de formatare", arg0, arg1, ..., argN);
```

În această versiune, parametrii lui `WriteLine()` sunt separați prin virgule în locul semnelor plus. *Șirul de formatare* conține două elemente: caractere afișabile obișnuite care sunt afișate fără modificări și specificatori de format. Specificatorii de format au forma generală:

{nr_arg, width: fmt}

Aici, *nr_arg* precizează numărul parametrului (începând cu zero) care trebuie afișat. Lățimea minimă a câmpului este stabilită prin valoarea *width*, iar formatul este specificat de către *fmt*.

La execuție, atunci când în șirul de formatare este întâlnit un specificator de format, parametrul corespunzător, precizat de către *nr_arg*, este substituit și afișat. În concluzie, poziția unei specificații de format în șirul de formatare este cea care determină unde se va afișa informația care îi corespunde. Atât *width*, cât și *fmt* sunt opționale. În cea mai simplă formă, un specificator de format indică deci parametrul care va fi afișat. De exemplu, {0} corespunde cu *arg0*, {1} cu *arg1*, ș.a.m.d.

Să începem cu un exemplu simplu. Instrucțiunea:

```
Console.WriteLine("Februarie are {0} sau {1} de zile.", 28, 29);
```

determină afișarea textului:

```
Februarie are 28 sau 29 de zile.
```

După cum observați, valoarea 28 este substituită în locul lui {0}, iar 29 în locul lui {1}. Specificatorii de format identifică deci poziția în care parametrii care urmează, în acest caz 28 și 29, sunt afișați în cadrul șirului. Mai mult, observați că valorile suplimentare sunt separate prin virgule, nu prin semne +.

Iată în continuare o formă ușor diferită a instrucțiunii anterioare care stabilește și lățimile minime ale câmpurilor:

```
Console.WriteLine("Februarie are {0, 10} sau {1, 5} de zile.", 28, 29);
```

Aceasta determină afișarea textului:

```
Februarie are         28 sau    29 de zile.
```

După cum se observă, s-au adăugat spații pentru umplerea porțiunilor neutilizate ale câmpurilor. Rețineți că este vorba despre lățimea *minimă* pentru fiecare câmp în parte. Textul afișat poate depăși această lățime, dacă este cazul.

În exemplele anterioare, valorile propriu-zise nu au fost formatate în nici un fel. Desigur însă că avantajul utilizării specificatorilor de format constă în controlul asupra aspectului datelor. Tipurile de date cel mai frecvent formatate sunt valorile în

virgulă mobilă și cele de tip `decimal`. Unul dintre cele mai simple moduri de a specifica un format este descriind direct șablonul pe care `WriteLine()` îl va aplica. Pentru aceasta, precizați un model al formatului dorit, utilizând semnul # pentru a marca pozițiile numerice. De exemplu, iată o modalitate îmbunătățită de afișare a rezultatului împărțirii lui 10 la 3:

```
Console.WriteLine("Valoarea lui 10/3: {0:#.##}", 10.0/3.0);
```

Această instrucțiune va afișa rezultatul prezentat mai jos:

```
Valoarea lui 10/3: 3.33
```

În acest exemplu, șablonul este #.##, ceea ce informează metoda `WriteLine()` să afișeze două zecimale. Este bine însă să rețineți că `WriteLine()` va afișa mai mult de o poziție la stânga punctului zecimal dacă este necesar, pentru a nu trunchia valoarea.

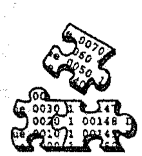
Pentru afișarea unor valori reprezentând sume de bani în formatul cu dolari și cenți, utilizați specificatorul de format `C`. De exemplu,

```
decimal sold;

sold = 12323.09m;
Console.WriteLine("Soldul curent este {0:C}", sold);
```

Această secvență va afișa următorul rezultat:

```
Soldul curent este $12,323.09
```



Proiectul 2-1: O convorbire cu planeta Marte

Planeta Marte se apropie în cel mai bun caz la o distanță de aproximativ 55 de milioane de kilometri de Pământ. Presupunând că pe Marte se găsește cineva cu care doriți să vorbiți, care este întârzierea dintre momentul în care o undă radio pleacă de pe Pământ și momentul în care aceeași undă ajunge pe Marte ? În cadrul acestui proiect, veți crea un program care răspunde la această întrebare. Reamintiți-vă că viteza luminii este de aproximativ 300 000 de kilometri pe secundă. Prin urmare, pentru a calcula întârzierea trebuie să împărțiți distanța la viteza luminii. Rezultatul va fi afișat în minute și secunde.

Pas cu pas

1. Creați un fișier nou, numit `Mars.cs`.
2. Pentru calculul întârzierii, este nevoie să utilizăm valori în virgulă mobilă. Aceasta deoarece valoarea intervalului are și o parte fracționară. Iată care sunt variabilele pe care programul le utilizează:

```
double distance;
double lightspeed;
double delay;
double delay_in_min;
```

3. Inițializăm variabilele `distance` și `lightspeed`, după cum urmează:

```
distance = 55000000;   // 55 000 000 kilometri
lightspeed = 300000;   // 300 000 km pe secunda
```

4. Pentru calculul întârzierii, **distance** se împarte la **lightspeed**. Rezultatul acestei operații este valoarea în secunde a întârzierii. Această valoare se atribuie variabilei **delay**, după care se afișează rezultatul. Acești pași sunt cuprinși în secvența:

```
delay = distance / lightspeed;
Console.WriteLine("Intarzierea la convorbirea cu Marte: " +
                  delay + " secunde.");
```

5. Se împarte numărul de secunde obținut anterior la 60, pentru a obține valoarea întârzierii în minute; acest rezultat se afișează utilizând secvența de cod:

```
delay_in_min = delay / 60;
Console.WriteLine("Intarzierea este de " + delay_in_min +
                  " minute.");
```

Iată în continuare listingul complet al programului **Mars.cs**:

```
/*
   Proiectul 2-1
   O convorbire cu planeta Marte

   Denumiti fisierul Mars.cs
*/

using System;

class Mars {
  static void Main() {
    double distance;
    double lightspeed;
    double delay;
    double delay_in_min;

    distance = 55000000;   // 55 000 000 kilometri
    lightspeed = 300000;   // 300 000 km pe secunda

    delay = distance / lightspeed;
    Console.WriteLine("Intarzierea la convorbirea cu Marte: " +
                      delay + " secunde.");

    delay_in_min = delay / 60;

    Console.WriteLine("Intarzierea este de " + delay_in_min +
                      " minute.");
  }
}
```

6. Compilați și rulați programul. Acesta va afișa următorul rezultat:

```
Intarzierea la convorbirea cu Marte: 183.333333333333 secunde.
Intarzierea este de 3.05555555555556 minute.
```

7. Pentru majoritatea celor interesați, programul afișează mult prea multe poziții zecimale. Pentru a face rezultatul mai ușor de citit, înlocuiți instrucțiunile **WriteLine()** din program cu cele de mai jos:

```
Console.WriteLine("Intarzierea la convorbirea cu Marte:
{0:###.###} secunde", delay);
Console.WriteLine("Intarzierea este de {0:###.###} minute",
delay_in_min);
```


8. Compilați și rulați încă o dată programul. Acesta va afișa rezultatul:

```
Intarzierea la convorbirea cu Marte: 183.333 secunde
Intarzierea este de 3.056 minute
```

Acum, acesta afișează doar trei poziții zecimale.

Literali

În C#, termenul de *literali* desemnează valorile fixate, reprezentate într-un format accesibil omului. De exemplu, numărul 100 este un literal. Literalii sunt denumiți în mod uzual și *constante*. În marea majoritate a cazurilor, construcția și utilizarea literalilor sunt atât de ușor de intuit, încât aceștia au fost utilizați într-o formă sau alta de toate programele prezentate anterior ca exemple. A sosit acum momentul să prezentăm și o explicație riguroasă.

Literalii din C# pot fi de orice tip valoric. Modul în care fiecare literal este reprezentat depinde de tipul său. După cum am precizat mai devreme, constantele de tip caracter sunt incluse între apostrofuri. De exemplu, 'a' și '%' reprezintă ambele constante de tip caracter.

Literalii întregi sunt specificați ca numere fără parte fracționară. De exemplu, 10 și –100 sunt constante întregi. Constantele în virgulă mobilă cer utilizarea punctului zecimal, urmat de partea fracționară a numărului. Spre exemplu, 11.123 este o constantă în virgulă mobilă. Limbajul C# permite de asemenea utilizarea notației științifice pentru numerele în virgulă mobilă.

Deoarece limbajul C# este puternic tipizat, și literalii aparțin unui tip. În mod firesc, aceasta dă naștere următoarei întrebări: Care este tipul unui literal numeric ? De exemplu, care este tipul pentru 12, 123987 sau 0.23 ? Din fericire, limbajul C# precizează câteva reguli ușor de aplicat pentru a răspunde acestor întrebări.

Mai întâi, pentru literalii întregi, tipul fiecărui literal este cel mai mic tip întreg care permite memorarea sa, începând de la **int**. În concluzie, un literal întreg poate fi de tipul **int**, **uint**, **long** sau **ulong**, în funcție de valoarea sa. Literalii în virgulă mobilă au implicit tipul **double**.

Dacă nu vă convine tipul pe care limbajul C# îl folosește implicit pentru un literal, puteți preciza explicit tipul adăugând un sufix. Pentru specificarea unui literal de tip **long**, se adaugă un *l* sau *L*. De exemplu, 12 este un **int**, în timp ce 12L este **long**. Pentru a specifica o valoare întreagă fără semn, se adaugă sufixul *u* sau *U*. Prin urmare, 100 este un **int**, dar 100U este de tip **uint**. Pentru specificarea unui întreg lung fără semn, se utilizează *ul* sau *UL*. Spre exemplu, valoarea 984375UL este de tipul **ulong**.

Pentru a specifica un literal de tipul **float**, se adaugă un *F* sau *f* la constantă. De exemplu, 10.19F este de tipul **float**.

Pentru specificarea unui literal de tip **decimal**, se adaugă la valoarea acestuia un *m* sau *M*. Spre exemplu, 9.95M este un literal de tip **decimal**.

Chiar dacă literalii întregi creează în mod implicit valori de tipul **int**, **uint**, **long** sau **ulong**, acestea pot fi atribuite unor variabile de tipul **byte**, **sbyte**, **short** sau **ushort** atât timp cât se pot reprezenta în cadrul tipului țintă. Un literal întreg poate fi întotdeauna atribuit unei variabile de tip **long**.

Literali hexazecimali

După cum probabil știți, în programare este uneori mai simplu să utilizăm sistemul de numerație în baza 16 decât cel în baza 10. Sistemul de numerație în baza 16 este denumit *hexazecimal* și utilizează cifrele de la 0 la 9 și literele de la A la F, care înlocuiesc valorile 10, 11, 12, 13, 14 și 15. De exemplu, numărul hexazecimal 10 are în zecimal valoarea 16. Datorită frecvenței mari de utilizare a numerelor hexazecimale, limbajul C# permite specificarea constantelor întregi în format hexazecimal. Orice literal hexazecimal trebuie să înceapă cu 0x (zero urmat de x). Iată câteva exemple:

```
count = 0xFF;     // 255 în zecimal
incr = 0x1a;      // 26 în zecimal
```

Secvențe escape pentru caractere

Includerea constantelor de tip caracter între apostrofuri funcționează pentru majoritatea caracterelor afișabile. Câteva caractere însă, cum ar fi cel de revenire la cap de linie (Carriage Return), ridică probleme deosebite atunci când se utilizează editoare de text pentru introducerea lor. Mai mult, alte câteva caractere, cum sunt apostrofurile și ghilimelele, au semnificații speciale în C#, deci nu pot fi utilizate în mod direct. Din aceste motive, C# pune la dispoziție *secvențe escape* speciale, denumite uneori și *constante caracter cu backslash*, prezentate în tabelul 2-2. Aceste secvențe sunt utilizate în locul caracterelor pe care le reprezintă.

Spre exemplu, instrucțiunea următoare atribuie lui **ch** caracterul tab:

```
ch = '\t';
```

Următorul exemplu atribuie tot lui **ch** un apostrof:

```
ch = '\'';
```

Tabelul 2-2 Secvențe escape

Secvența escape	Descriere
\a	alarmă
\b	șterge un caracter în urmă (backspace)
\f	form feed
\n	linie nouă (linefeed)
\r	revenire la cap de rând (carriage return)
\t	tab orizontal
\v	tab vertical
\0	nul
\'	apostrof
\"	ghilimele
\\	backslash

Literali de tipul string

În C#, putem întâlni și un alt tip de literali: tipul string. Un *string* reprezintă un șir de caractere inclus între ghilimele. De exemplu,

```
"acesta este un test"
```

este un șir de caractere. Ați observat exemple de șiruri de caractere în multe din instrucțiunile **WriteLine()** din programele prezentate ca exemple până acum.

Pe lângă caractere obișnuite, un literal de tipul șir de caractere poate conține și una sau mai multe din secvențele escape prezentate mai sus. De exemplu, să considerăm programul din continuare. Acesta utilizează secvențele escape **\n** și **\t**.

```
// Prezinta utilizarea secventelor escape in sirurile de caractere

using System;

class StrDemo {
  static void Main() {
    Console.WriteLine("Prima linie\nA doua linie");
    Console.WriteLine("A\tB\tC");
    Console.WriteLine("D\tE\tF");
  }
}
```

Utilizarea \n pentru a trece la linie nouă

Utilizarea tab-urilor pentru aliniere

Iată ce afișează programul:

```
Prima linie
A doua linie
A    B    C
D    E    F
```

Sfatul expertului

Întrebare: Știu că limbajul C++ permite ca literalii întregi să fie specificați în octal (sistem de numerație în baza 8). Se întâmplă aceasta și în C# ?

Răspuns: Nu. Limbajul C# permite specificarea literalilor numai în format zecimal sau hexazecimal. Sistemul octal este foarte rar utilizat în mediile de programare moderne.

Observați utilizarea secvenței escape `\n` pentru trecerea la o linie nouă. Nu este nevoie să utilizați mai multe instrucțiuni `WriteLine()` pentru a realiza o afișare pe mai multe linii. Este suficient să introduceți secvența `\n` în interiorul unui șir lung de caractere, în locurile unde doriți efectuarea trecerii la o linie nouă.

Pe lângă literalii de tip șir de caractere prezentați până acum, se mai pot de asemenea specifica literali *„copie la indigo"*. Un astfel de literal începe cu @, urmat de un șir cuprins între ghilimele. Conținutul șirului cuprins între ghilimele este acceptat fără modificări și se poate întinde pe mai multe linii. Puteți deci include caractere linie nouă, tab, etc. fără să fie nevoie să utilizați secvențe escape. Singura excepție este că, pentru a obține ghilimelele ("), trebuie să utilizați două astfel de caractere unul după altul (""). Iată un program care prezintă literalii „copie la indigo":

```
// Prezentarea literalilor "copie la indigo"

using System;

class Indigo {
  static void Main() {
    Console.WriteLine(@"Acesta este un literal
copie la indigo
care ocupa mai multe linii.
");
    Console.WriteLine(@"Iata niste rezultate afisate tabelar:
1    2    3    4
5    6    7    8
");
    Console.WriteLine(@"Programatorii afirma: ""Ne place C#.""");
  }
}
```

Acest literal „copie la indigo" conține caractere de trecere la linie nouă...

...iar acesta conține caractere tab

Programul afișează textul prezentat în continuare:

```
Acesta este un literal
copie la indigo
care ocupa mai multe linii.

Iata niste rezultate afisate tabelar:
1    2    3    4
5    6    7    8

Programatorii afirma: "Ne place C#."
```

Ceea ce este important de observat în programul anterior este că literalii „copie la indigo" sunt afișați exact la fel cum sunt introduși în textul programului.

Avantajul utilizării literalilor „copie la indigo" este că puteți descrie modul de afișare în program exact la fel cum va apărea pe ecran. În cazul șirurilor multilinie, trecerea pe o altă linie va ascunde indentarea programului. Din acest motiv, programele din această carte nu vor utiliza literali „copie la indigo". Cu toate acestea, ei reprezintă un avantaj prețios în multe situații cu dificultăți la formatare.

Exerciții la minut

- Care este tipul literalului 10 ? Dar al literalului 10.0 ?
- Cum specificați 100 ca `long` ? Dar ca `uint` ?
- Ce reprezintă @„test" ?

Sfatul expertului

Întrebare: Șirul cu un singur caracter reprezintă același lucru cu un literal de tip caracter ? De exemplu, „k" este identic cu 'k' ?

Răspuns: Nu. Nu trebuie să confundați șirurile de caractere cu caracterele simple. Un literal de tip caracter reprezintă o singură literă de tipul `char`. Un șir de caractere, chiar dacă are o singură literă, rămâne totuși un obiect diferit. Chiar dacă șirurile sunt alcătuite din caractere, ele au un tip diferit.

O privire amănunțită asupra variabilelor

Variabilele au fost prezentate în modulul 1. După cum ați învățat atunci, variabilele sunt declarate printr-o instrucțiune de forma:

tip nume-var;

unde *tip* reprezintă tipul variabilei, iar *nume-var* este numele său. Puteți declara variabile având orice tip valid, inclusiv tipurile valorice discutate până acum. Atunci când creați o variabilă, creați de fapt o instanță a tipului său. Operațiile permise asupra unei variabile sunt deci determinate de tipul acesteia. De exemplu, nu puteți utiliza o variabilă de tipul `bool` pentru a memora valori în virgulă mobilă. Mai mult, tipul unei variabile nu se poate modifica pe parcursul duratei sale de viață. Spre exemplu, o variabilă de tipul `int` nu se poate transforma într-una de tipul `char`.

În C#, toate variabilele trebuie declarate înainte de a fi folosite. Aceasta este necesar deoarece compilatorul trebuie să cunoască tipul de date conținut de o variabilă înainte de a compila orice instrucțiune care o utilizează. Aceasta permite limbajului C# să efectueze verificarea strictă a tipurilor.

În C#, sunt definite tipuri diferite de variabile. Cele pe care deja le-am utilizat se numesc *variabile locale*, deoarece sunt declarate în interiorul unei metode.

- 10 este de tipul `int`, iar 10.0 este `double`.
- 100 ca `long` este 100L. 100 ca `uint` este 100U.
- @„test" este un literal „copie la indigo".

Inițializarea unei variabile

Trebuie să atribuiți unei variabile o valoare înainte de a o utiliza. O modalitate de a realiza aceasta este prin intermediul unei instrucțiuni de atribuire, după cum ați văzut deja. O alta este de a atribui o valoare inițială atunci când variabila este declarată. Pentru aceasta, după numele variabilei se scriu semnul egal și valoarea care îi este atribuită. Formatul general al inițializării este prezentat mai jos:

tip var = val;

Aici, *val* reprezintă valoarea atribuită variabilei *var* atunci când aceasta este creată. Valoarea trebuie să fie compatibilă cu tipul precizat.

Iată câteva exemple:

```
int count = 10;  // count primeste valoarea initiala 10
char ch = 'X';   // initializam ch cu litera X
float f = 1.2F;  // f este initializata cu 1.2
```

La declararea a două sau mai multe variabile de același tip utilizând o listă separată prin virgule, puteți inițializa una sau mai multe dintre variabile. De exemplu:

```
int a, b = 8, c = 19, d;  // b si c sunt initializate
```

În acest caz, numai **b** și **c** sunt inițializate.

Inițializarea dinamică

Deși în exemplele anterioare am folosit numai constante ca inițializatori, limbajul C# permite ca variabilele să fie inițializate dinamic, utilizând orice expresie validă la momentul în care variabila este declarată. Spre exemplu, iată un program de câteva linii care calculează volumul unui cilindru cunoscând raza bazei sale și înălțimea:

```
// Prezentarea initializarii dinamice

using System;

class DynInit {
  static void Main() {
    double radius = 4, height = 5;

    // initializarea dinamica a volumului
    double volume = 3.1416 * radius * radius * height;

    Console.WriteLine("Volumul este " + volume);
  }
}
```

volume este inițializată dinamic la rulare

În program sunt declarate trei variabile locale – **radius**, **height** și **volume**. Primele două, **radius** (raza) și **height** (înălțimea), sunt inițializate cu valori constante. Variabila **volume** este însă inițializată dinamic cu volumul cilindrului. Esențialul este că expresia de inițializare poate utiliza orice element valabil la momentul inițializării, inclusiv apeluri de metode, alte variabile sau literali.

Domeniul de valabilitate și durata de viață a variabilelor

Până acum, am declarat toate variabilele pe care le-am utilizat la începutul metodei **Main()**. Limbajul C# permite însă declararea unei variabile locale în interiorul oricărui bloc. După cum am precizat în modulul 1, un bloc începe cu o acoladă deschisă și se termină cu o acoladă închisă. Un bloc delimitează un *spațiu de declarare*, numit și *domeniu de valabilitate*. De fiecare dată când începeți un bloc nou, creați de fapt un nou domeniu de valabilitate. Un domeniu de valabilitate stabilește care obiecte sunt vizibile în alte părți ale programului. De asemenea, acesta determină și durata de viață a acelor obiecte.

Cele mai importante domenii de valabilitate în C# sunt acelea definite de o clasă și respectiv de o metodă. Despre domeniul de valabilitate asociat unei clase (și despre variabilele declarate în acest domeniu) vom discuta ulterior în această carte, când vom prezenta noțiunea de clasă. Ne limităm acum la studiul domeniilor de valabilitate definite de către o metodă sau în interiorul acesteia.

Domeniul de valabilitate definit de o metodă începe o dată cu acolada de deschidere. Dacă însă acea metodă are parametri, aceștia sunt la rândul lor incluși în domeniul metodei.

Ca regulă generală, variabilele declarate în interiorul unui domeniu nu sunt vizibile (adică accesibile) pentru codul definit în afara acelui domeniu. Prin urmare, declararea unei variabile într-un domeniu localizează acea variabilă, protejând-o împotriva accesului și modificării neautorizate. Regulile de accesibilitate pentru un domeniu constituie astfel un fundament pentru încapsulare.

Domeniile de valabilitate pot fi imbricate. De exemplu, de fiecare dată când creați un bloc de cod, creați de fapt și un nou domeniu imbricat. În acest caz, domeniul exterior îl conține pe cel interior. Aceasta înseamnă că obiectele declarate în domeniul exterior sunt vizibile pentru codul din domeniul interior. Reciproca nu este însă adevărată. Obiectele declarate în domeniul interior nu rămân vizibile în afara acestuia.

Pentru a înțelege efectul domeniilor imbricate, să considerăm următorul program:

```
// Prezentarea domeniilor de valabilitate

using System;

class ScopeDemo {
  static void Main() {
    int x;  // cunoscuta in toata metoda Main()

    x = 10;
    if(x == 10) { // incepem un nou domeniu

      int y = 20;
```

```
      // x si y sunt ambele cunoscute
      Console.WriteLine("x si y: " + x + " " + y);
      x = y * 2;
    }

    // y = 100; // Eroare ! y nu este cunoscut aici

    // x inca este cunoscut aici.

    Console.WriteLine("x este " + x);
  }
}
```

y se găseşte în afara domeniului său.

Aşa cum ne arată comentariile, variabila **x** este declarată la începutul domeniului metodei **Main()**, fiind accesibilă pentru întreg codul adăugat ulterior în corpul metodei. În interiorul blocului **if**, se declară variabila **y**. Deoarece un bloc defineşte un domeniu, **y** este vizibilă numai pentru codul aflat în cadrul blocului său. Din acest motiv am comentat linia **y = 100;** ce se găseşte în afara blocului. Dacă veţi elimina marcajul de comentariu din faţa acestei linii, se va produce o eroare la compilare, deoarece **y** nu este vizibilă în afara blocului său. În cadrul blocului **if**, variabila **x** poate fi folosită deoarece codul dintr-un bloc (care este un domeniu imbricat) are acces la variabilele declarate în domeniul care îl cuprinde.

În cadrul unui bloc, variabilele pot fi declarate în orice punct, dar sunt valide numai după declaraţie. În concluzie, dacă declaraţi o variabilă la începutul unei metode, aceasta este disponibilă pe întreg parcursul metodei. Reciproc, dacă declaraţi o variabilă la sfârşitul unui bloc, aceasta este practic inutilă, deoarece nu există nici o porţiune de cod în care să fie accesibilă.

Iată un alt amănunt important de reţinut: variabilele se creează atunci când fluxul de execuţie intră în domeniul lor, fiind distruse atunci când domeniul este părăsit. Aceasta înseamnă că o variabilă nu îşi mai păstrează valoarea după ce domeniul său a fost părăsit. În consecinţă, variabilele declarate într-o metodă nu îşi păstrează valorile din apelurile anterioare ale aceleiaşi metode. De asemenea, o variabilă declarată în interiorul unui bloc îşi pierde valoarea atunci când blocul este părăsit. Durata de viaţă a unei variabile este deci restrânsă la domeniul său de valabilitate.

Dacă o declaraţie de variabilă conţine un iniţializator, atunci variabila va fi reiniţializată de fiecare dată când se intră în blocul în care este declarată. De exemplu, să considerăm programul:

```
// Prezentarea duratei de viata a unei variabile.

using System;

class VarInitDemo {
  static void Main() {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1;  // y se initializeaza la fiecare intrare in bloc
```



```
      Console.WriteLine("y este: " + y);  // afiseaza intotdeauna -1
      y = 100;
      Console.WriteLine("y este acum: " + y);
    }
  }
}
```

Programul va afişa textul prezentat mai jos:

```
y este: -1
y este acum: 100
y este: -1
y este acum: 100
y este: -1
y este acum: 100
```

După cum puteţi observa, **y** este reiniţializată cu –1 de fiecare dată când se intră în bucla interioară **for**. Chiar dacă ulterior i se atribuie valoarea 100, aceasta se pierde.

Există o ciudăţenie în ceea ce priveşte regulile de domeniu în C# care vă poate surprinde: deşi blocurile pot fi imbricate, nici o variabilă declarată într-un domeniu interior nu poate avea acelaşi nume cu o altă variabilă, declarată într-un domeniu care îl conţine. De exemplu, următorul program, care încearcă declararea a două variabile diferite cu acelaşi nume, nu se va compila:

```
/*
   Acest program incearca sa declare o variabila
   intr-un domeniu interior cu acelasi nume cu alta
   variabila definita intr-un domeniu exterior.

   *** Acest program nu se poate compila. ***
*/

using System;

class NestVar {
  static void Main() {
    int count;

    for(count = 0; count < 10; count = count + 1) {
      Console.WriteLine("Valoarea lui count: " + count);

      int count;    // declaratie nepermisa !!!
      for(count = 0; count < 2; count++)
        Console.WriteLine("Acest program contine erori !");
    }
  }
}
```

Nu putem declara din nou pe **count**, fiind deja declarată în **Main()**

Dacă aţi lucrat anterior în C sau C++, ştiţi că acolo nu există restricţii asupra numelor variabilelor declarate într-un domeniu interior. Prin urmare, în C/C++, redeclararea variabilei **count** în cadrul blocului buclei **for** exterioare este perfect valabilă. În C/C++, o astfel de declaraţie ascunde variabila exterioară. Cei care au proiectat limbajul C# au considerat că această *ascundere a numelor* poate conduce cu uşurinţă la erori de programare şi au interzis-o.

Exerciții la minut

- Ce este un domeniu de valabilitate ? Cum se poate crea un domeniu de valabilitate ?
- Unde se pot declara variabilele în cadrul unui bloc ?
- În cadrul unui bloc, când se creează o variabilă ? Când se distruge variabila creată ?

Operatori

Limbajul C# pune la dispoziție un set bogat de operatori. Un operator este un simbol care determină compilatorul să efectueze o anumită prelucrare matematică sau logică. În C#, avem patru categorii generale de operatori: *aritmetici*, *pe biți*, *relaționali* și *logici*. Există de asemenea în C# câțiva operatori suplimentari, necesari pentru rezolvarea unor situații speciale. În acest modul, vom examina operatorii aritmetici, relaționali și logici. Vom studia de asemenea și operatorul de atribuire. Operatorii pe biți și cei speciali vor fi examinați ulterior.

Operatori aritmetici

Limbajul C# definește următorii operatori aritmetici:

Operatorul	Semnificația
+	Adunare
-	Scădere (și minus unar)
*	Înmulțire
/	Împărțire
%	Rest (operatorul modulo)
++	Incrementare
--	Decrementare

Operatorii +, -, * și / funcționează în C# la fel ca în orice alt limbaj de programare (și totodată la fel ca în algebră). Acești operatori se pot aplica asupra oricăror tipuri numerice predefinite.

Chiar dacă acțiunea operatorilor aritmetici este binecunoscută tuturor cititorilor, există câteva situații speciale care justifică unele explicații. Mai întâi, să ne amintim că atunci când / se aplică asupra unor întregi, restul este trunchiat; de exemplu, 10/3 este egal cu 3 la împărțirea întreagă. Restul la împărțirea întreagă se poate

- Un domeniu de valabilitate definește vizibilitatea și durata de viață a obiectelor. Orice bloc definește un domeniu de valabilitate.
- Variabilele se pot declara în orice punct în cadrul unui bloc.
- O variabilă este creată la întâlnirea declarației sale în cadrul blocului. Variabila este distrusă la părăsirea blocului.

obține utilizând operatorul modulo %. Acesta funcționează și în C# la fel ca în alte limbaje: întoarce restul unei împărțiri întregi. De exemplu, 10 % 3 este 1. În C#, operatorul % se poate aplica atât tipurilor întregi, cât și celor în virgulă mobilă. În consecință, 10.0 % 3.0 este egal tot cu 1. (Aceasta este o diferență față de C și C++, care permit operații modulo numai asupra tipurilor întregi.) Următorul program prezintă funcționarea operatorului modulo:

```
// Prezentarea operatorului %.

using System;

class ModDemo {
  static void Main() {
    int iresult, irem;
    double dresult, drem;

    iresult = 10 / 3;
    irem = 10 % 3;

    dresult = 10.0 / 3.0;
    drem = 10.0 % 3.0;

    Console.WriteLine("Catul si restul impartirii 10 / 3: " +
                      iresult + " " + irem);
    Console.WriteLine("Catul si restul impartirii 10.0 / 3.0: " +
                      dresult + " " + drem);
  }
}
```

Programul afișează rezultatul prezentat mai jos:

```
Catul si restul impartirii 10 / 3: 3 1
Catul si restul impartirii 10.0 / 3.0: 3.33333333333333 1
```

După cum se observă, operatorul % întoarce 1 atât la împărțirea întreagă, cât și la operația cu numere în virgulă mobilă.

Incrementarea și decrementarea

Prezentați în modulul 1, ++ și - - sunt operatorii de incrementare, respectiv de decrementare. După cum vom vedea, aceștia prezintă unele proprietăți speciale, care îi fac destul de interesanți. Să începem prin a recapitula exact ce fac operatorii de incrementare și decrementare.

Operatorul de incrementare adaugă 1 operandului său, în timp ce operatorul de decrementare scade 1. În consecință,

```
x = x + 1;
```

este același lucru cu

```
x++;
```

iar

```
x = x - 1;
```

este echivalentă cu

```
--x;
```

Atât operatorul de incrementare, cât şi cel de decrementare îşi pot precede (în forma *prefixată*) sau urma (în forma *postfixată*) operandul. Spre exemplu:

```
x = x + 1;
```

se poate rescrie ca:

```
++x;   // forma prefixata
```

sau ca:

```
x++;   // forma postfixata
```

În exemplul anterior, nu apare nici o diferenţă dacă incrementarea se aplică sub formă prefixată sau postfixată. Atunci când însă incrementarea sau decrementarea face parte dintr-o expresie mai mare, există o diferenţă importantă. Dacă operatorul de incrementare sau de decrementare îşi *precede* operandul, operaţia se va efectua înainte de a calcula valoarea operandului care va fi utilizată în restul expresiei. Dacă însă operatorul *urmează* operandului, compilatorul va obţine valoarea operandului înainte de a-l incrementa sau decrementa. Să considerăm secvenţa:

```
x = 10;
y = ++x;
```

În acest caz, y va primi valoarea 11. Dacă însă rescriem codul sub forma:

```
x = 10;
y = x++;
```

atunci y va primi valoarea 10. În ambele cazuri, x primeşte valoarea 11; diferă însă momentul când aceasta se întâmplă. Controlul asupra momentului când se execută operaţiile de incrementare sau decrementare aduce uneori avantaje importante.

Operatori relaţionali şi logici

În sintagmele *„operatori relaţionali"* şi *„operatori logici"*, *relaţional* se referă la relaţiile de ordine care pot exista între două valori, iar *logic* desemnează modalităţile în care se pot asocia valorile de adevăr true şi false. Deoarece operatorii relaţionali produc rezultate adevărate sau false, aceştia se utilizează deseori împreună cu cei logici. Acesta este şi motivul pentru care îi prezentăm aici împreună.

Operatorii relaţionali sunt prezentaţi mai jos:

Operatorul	Semnificaţia
==	Egal cu
!=	Diferit de
>	Mai mare decât
<	Mai mic decât
>=	Mai mare sau egal cu
<=	Mai mic sau egal cu

În continuare, prezentăm şi operatorii logici:

Operatorul	Semnificaţia
&	ŞI
|	SAU
^	SAU exclusiv (XOR)
||	SAU „scurtcircuitat"
&&	ŞI „scurtcircuitat"
!	NON

Rezultatele întoarse de operatorii relaţionali şi logici sunt valori de tipul **bool**.

În C#, toate obiectele pot fi comparate în ceea ce priveşte egalitatea sau neegalitatea, utilizând operatorii == şi !=. Operatorii de comparaţie <, >, <= şi >= se pot aplica însă numai tipurilor pe care este definită o relaţie de ordine. În concluzie, toţi operatorii relaţionali pot fi aplicaţi tuturor tipurilor numerice. Valorile de tipul **bool** pot fi însă comparate numai în ce priveşte egalitatea sau neegalitatea, deoarece valorile **true** şi **false** nu sunt ordonate. Relaţia **true > false** nu are deci nici o semnificaţie în C#.

Pentru operatorii logici, operanzii trebuie să fie de tipul **bool**, rezultatul operaţiei logice fiind tot de tipul **bool**. Operatorii logici &, |, ^ şi ! implementează operaţiile logice elementare ŞI, SAU, SAU exclusiv (XOR) şi NON, în conformitate cu următoarele tabele de adevăr:

p	q	p & q	p | q	p ^ q	! p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

După cum se observă din tabelă, rezultatul operaţiei SAU exclusiv este true atunci când unul şi numai unul dintre operanzi este true.

În continuare, iată un program care prezintă câţiva dintre operatorii relaţionali şi logici:

```
// Prezentarea operatorilor relationali si logici.

using System;

class RelLogOps {
  static void Main() {
    int i, j;
    bool b1, b2;

    i = 10;
    j = 11;
    if(i < j) Console.WriteLine("i < j");
```

```
    if(i <= j) Console.WriteLine("i <= j");
    if(i != j) Console.WriteLine("i != j");
    if(i == j) Console.WriteLine("Nu se va executa");
    if(i >= j) Console.WriteLine("Nu se va executa");
    if(i > j) Console.WriteLine("Nu se va executa");

    b1 = true;
    b2 = false;

    if(b1 & b2) Console.WriteLine("Nu se va executa");
    if(!(b1 & b2)) Console.WriteLine("!(b1 & b2) este true");
    if(b1 | b2) Console.WriteLine("b1 | b2 este true");
    if(b1 ^ b2) Console.WriteLine("b1 ^ b2 este true");
  }
}
```

Textul afişat de program este prezentat mai jos:

```
i < j
i <= j
i != j
!(b1 & b2) este true
b1 | b2 este true
b1 ^ b2 este true
```

Operatori logici „scurtcircuitaţi"

În C#, dispunem şi de variante „*scurtcircuitate*" ale operatorilor logici ŞI, respectiv SAU, care pot fi utilizate pentru a scrie programe mai eficiente. Pentru a înţelege de ce sunt aceste variante mai eficiente, să considerăm următoarea situaţie. Într-o operaţie ŞI, dacă primul operand este fals, rezultatul va fi fals indiferent de valoarea celui de-al doilea operand. Pentru o operaţie SAU, dacă primul operand este adevărat, atunci rezultatul operaţiei este adevărat indiferent de ce valoare are al doilea operand. În aceste două cazuri, nu este deci necesar să evaluăm al doilea operand. Neevaluând al doilea operand, se câştigă timp, iar codul rezultat este mai eficient.

Operatorii „scurtcircuitaţi" ŞI şi SAU sunt &&, respectiv ||. După cum am văzut mai devreme, aceştia au ca pereche operatorii obişnuiţi &, respectiv |. Singura diferenţă dintre variantele obişnuite şi cele scurtcircuitate este că operatorii obişnuiţi evaluează întotdeauna ambii operanzi, în timp ce variantele scurtcircuitate evaluează al doilea operand numai dacă este necesar.

Iată în continuare un program care prezintă operatorul ŞI scurtcircuitat. Programul stabileşte dacă valoarea memorată în `d` este un divizor al lui `n`. Aceasta se realizează efectuând o operaţie modulo. Dacă restul împărţirii `n / d` este zero, atunci `d` este un divizor. Cum însă operaţia modulo implică o împărţire, vom utiliza forma scurtcircuitată a operatorului ŞI pentru a evita o eroare de împărţire la zero.

```
// Prezinta operatorii scurtcircuitati

using System;
```



```
class Scops {
  static void Main() {
    int n, d;

    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(d + " este divizor al lui " + n);

    d = 0; // il facem pe d egal cu zero

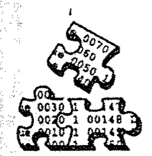
    // Cum d este zero, al doilea operand nu se evalueaza.
    if(d != 0 && (n % d) == 0)
      Console.WriteLine(d + " este divizor al lui " + n);

    /* Rescriem testul fara operatorul scurtcircuitat.
       Se va produce o eroare de impartire la zero.
    */

    if(d != 0 & (n % d) == 0)
      Console.WriteLine(d + " este divizor al lui " + n);
  }
}
```

Pentru a evita eroarea de împărţire la zero, instrucţiunea `if` verifică mai întâi dacă `d` este zero. În caz afirmativ, ŞI-ul scurtcircuitat se va opri fără să mai efectueze împărţirea. În primul test, deci, `d` are valoarea 2 şi operaţia modulo este efectuată. Al doilea test eşuează, deoarece `d` a primit valoarea zero, iar operaţia modulo este omisă, evitându-se eroarea de împărţire la zero. În fine, se încearcă utilizarea operatorului ŞI obişnuit. Aceasta determină evaluarea ambilor operanzi, ceea ce conduce la o eroare de execuţie atunci când se efectuează împărţirea la zero.

Operatorul ŞI scurtcircuitat se mai numeşte *ŞI condiţionat*, în timp ce operatorul SAU scurtcircuitat este denumit *SAU condiţionat*.



LogicOpTable.cs

Proiectul 2-2: Afişarea unei tabele de adevăr pentru operatorii logici

În cadrul acestui proiect, veţi crea un program care afişează tabelele de adevăr pentru operatorii logici din C#. Va trebui să aliniaţi corespunzător coloanele tabelului. Acest proiect face apel la câteva dintre facilităţile descrise în acest modul, printre care secvenţele escape şi operatorii logici. Programul evidenţiază de asemenea diferenţa de precedenţă între operatorul aritmetic + şi operatorii logici.

Pas cu pas

1. Creaţi un fişier nou, numit `LogicalOpTable.cs`.
2. Pentru a asigura alinierea coloanelor, utilizaţi secvenţa escape `\t` pentru a include tab-uri în fiecare şir de caractere afişat. Spre exemplu, următoarea instrucţiune `WriteLine()` afişează capul de tabel:

```
Console.WriteLine("P\tQ\tSI\tSAU\tXOR\tNON");
```

3. Pe liniile următoare din tabel, utilizați caractere tab pentru a poziționa rezultatul fiecărei operații pe coloana cu antetul corespunzător.
4. Iată listingul complet al programului **LogicalOpTable.cs**, pe care îl puteți introduce acum:

```
/*
   Proiectul 2-2

   Afisarea unei tabele de adevar pentru operatorii logici.
*/

using System;

class LogicalOpTable {
  static void Main() {

    bool p, q;

    Console.WriteLine("P\tQ\tSI\tSAU\tXOR\tNON");

    p = true; q = true;
    Console.Write(p + "\t" + q + "\t");
    Console.Write((p&q) + "\t" + (p|q) + "\t");
    Console.WriteLine((p^q) + "\t" + (!p));

    p = true; q = false;
    Console.Write(p + "\t" + q + "\t");
    Console.Write((p&q) + "\t" + (p|q) + "\t");
    Console.WriteLine((p^q) + "\t" + (!p));
    p = false; q = true;
    Console.Write(p + "\t" + q + "\t");
    Console.Write((p&q) + "\t" + (p|q) + "\t");
    Console.WriteLine((p^q) + "\t" + (!p));

    p = false; q = false;
    Console.Write(p + "\t" + q + "\t");
    Console.Write((p&q) + "\t" + (p|q) + "\t");
    Console.WriteLine((p^q) + "\t" + (!p));
  }
}
```

5. Compilați și rulați programul. Acesta va afișa următorul tabel:

```
P       Q       SI      SAU     XOR     NON
true    true    true    true    false   false
true    false   false   true    true    false
false   true    false   true    true    true
false   false   false   false   false   true
```

6. Remarcați parantezele care delimitează operațiile logice în instrucțiunile **Write()** și **WriteLine()** din program. Acestea sunt obligatorii din cauza precedenței operatorilor în C#. Operatorul + are precedență superioară operatorilor logici.
7. Încercați pe cont propriu să modificați programul astfel încât să utilizeze și să afișeze valorile 1 și 0 în loc de **true** și **false**. Aceasta presupune un efort mai mare decât ați putea crede inițial.

Sfatul expertului

Întrebare: Din moment ce operatorii scurtcircuitați sunt în majoritatea cazurilor mai eficienți decât operatorii obișnuiți echivalenți, de ce dispunem în C# de ambele variante de operatori ȘI și SAU ?

Răspuns: În unele cazuri, este necesar să evaluăm ambii operanzi ai unei operații ȘI ori SAU, datorită efectelor secundare produse. Să considerăm următorul program:

```
// Efectele secundare pot conta foarte mult.

using System;

class SideEffects {
  static void Main() {
    int i;

    i = 0;
    /* Aici, i este in continuare incrementat desi
       instructiunea if nu se executa. */
    if(false & (++i < 100))
       Console.WriteLine("Nu este afisata"); // se afiseaza 1
    Console.WriteLine("Dupa primul if i = " + i);

    /* Acum, i nu va mai fi incrementat din cauza operatorului
       scurtcircuitat care nu mai evalueaza al doilea operand. */
    if(false && (++i < 100))
      Console.WriteLine("Nu este afisata"); // se afiseaza tot 1 !
    Console.WriteLine("Dupa al doilea if i = " + i);
  }
}
```

După cum arată și comentariile, în prima instrucțiune **if**, **i** este incrementat chiar dacă instrucțiunea se execută sau nu. Atunci când folosim însă operatorul scurtcircuitat, variabila **i** nu mai este incrementată dacă primul operand este **false**. Rețineți de aici că dacă programul dumneavoastră așteaptă ca al doilea operand al unei operații ȘI ori SAU să fie evaluat, trebuie să utilizați formele nescurtcircuitate ale acestor operații.

Exerciții la minut

- Ce face operatorul % ? Cu ce tipuri lucrează el ?
- Ce tip de valori poate fi utilizat la operanzii operatorilor logici ?
- Operatorii scurtcircuitați își evaluează întotdeauna ambii operanzi ?

- % este operatorul modulo, care întoarce restul unei împărțiri întregi. Operatorul poate fi aplicat asupra tuturor tipurilor numerice.
- Operatorii logici lucrează doar cu operanzi de tipul **bool**.
- Nu, operatorii scurtcircuitați își evaluează al doilea operand numai dacă rezultatul operației nu poate fi stabilit sigur după evaluarea primului.

Operatorul de atribuire

Aţi mai utilizat operatorul de atribuire şi în modulul 1. A sosit acum momentul să îl studiem şi din punct de vedere formal. Operatorul de atribuire este reprezentat prin semnul egal, =. El funcţionează şi în C# aproape la fel ca în alte limbaje de programare. Formatul general al său este:

var = expr;

Tipul variabilei *var* trebuie să fie compatibil cu tipul expresiei *expr*.

Operatorul de atribuire are o proprietate interesantă pe care poate că nu o cunoaşteţi: vă permite să creaţi un şir de atribuiri. De exemplu, să considerăm secvenţa:

```
int x, y, z;

x = y = z = 100;  // initializam x, y si z cu 100
```

Secvenţa iniţializează variabilele **x**, **y** şi **z** cu 100 utilizând o singură instrucţiune. Aceasta este posibil deoarece operatorul = întoarce valoarea expresiei din membrul drept. Prin urmare, valoarea întoarsă de **z = 100** este 100, care este apoi atribuită lui y şi mai departe lui x. Utilizarea unui „şir de atribuiri" este o modalitate simplă de a da unui grup de variabile o valoare comună.

Atribuiri compuse

Limbajul C# pune la dispoziţie operatori compuşi de atribuire care simplifică scrierea anumitor instrucţiuni de atribuire. Să începem cu un exemplu. Instrucţiunea de atribuire de mai jos:

```
x = x + 10;
```

se poate rescrie utilizând o atribuire compusă astfel:

```
x += 10;
```

Operatorul compus += determină compilatorul să atribuie variabilei **x** valoarea anterioară a lui **x** plus 10.

Iată un alt exemplu. Instrucţiunea:

```
x = x - 100;
```

este echivalentă cu:

```
x -= 100;
```

Ambele instrucţiuni îi atribuie lui **x** valoarea sa anterioară minus 100.

Există operatori compuşi de atribuire pentru toţi operatorii binari (care necesită deci doi operanzi). Formatul general al atribuirii comprimate este:

var op = expr;

Operatorii compuşi de atribuire pentru operaţiile aritmetice şi logice sunt următorii:

+=	-=	*=	/=
%=	&=	|=	^=

Din cauză că operatorii compuşi de atribuire au o formă mai scurtă decât operatorii simpli echivalenţi cu ei, ei sunt uneori denumiţi şi operatori *comprimaţi* de atribuire.

Operatorii compuşi de atribuire au două avantaje. Mai întâi, sunt mai „compacţi" decât operatorii necomprimaţi echivalenţi cu ei. În al doilea rând, ei conduc la scrierea de programe mai eficiente (operanzii fiind evaluaţi o singură dată). Din aceste motive, veţi întâlni deseori operatorii compuşi de atribuire în programele C# „de firmă".

Conversia tipurilor la atribuire

O practică frecventă în programare este atribuirea unei variabile de un tip altei variabile având tipul diferit. Spre exemplu, se poate să doriţi atribuirea unei valori de tipul **int** unei variabile de tip **float**, ca în secvenţa următoare:

```
int i;
float f;

i = 10;
f = i;    // atribuim un int unui float
```

Dacă în atribuire sunt implicate tipuri compatibile, atunci valoarea din partea dreaptă este convertită automat la tipul din partea stângă. În secvenţa anterioară, valoarea lui **i** este mai întâi convertită la tipul **float** şi apoi atribuită lui **f**. Din cauza verificării stricte a tipurilor, nu toate tipurile din C# sunt compatibile şi implicit nu sunt permise toate conversiile de tip. De exemplu, tipurile **bool** şi **int** nu sunt compatibile.

Atunci când un tip de date este atribuit unui alt tip de variabilă, se efectuează o conversie *automată* de tip dacă:

- Cele două tipuri sunt compatibile.
- Tipul destinaţie este mai cuprinzător decât tipul sursă.

Dacă aceste două condiţii sunt îndeplinite, are loc o *conversie prin lărgire*. De exemplu, tipul **int** este suficient de cuprinzător pentru a memora orice valoare corectă de tip **byte**, iar atât **int** cât şi **byte** sunt tipuri întregi, deci se poate efectua o conversie automată.

Pentru conversiile prin lărgire, tipurile numerice, atât cele întregi cât şi cele în virgulă mobilă, sunt compatibile unele cu celelalte. De exemplu, programul următor este perfect corect, deoarece conversia de la **long** la **double** se efectuează automat prin lărgire:

```
// Prezinta conversia automata din long in double.

using System;
```

```
class LtoD {
  static void Main() {
    long L;
    double D;

    L = 100123285L;
    D = L;  ←  Conversie automată din long în double

    Console.WriteLine("L si D: " + L + " " + D);
  }
}
```

Chiar dacă se poate efectua conversia automată din **long** în **double**, operația inversă (conversia din **double** în **long**) nu se poate efectua automat, deoarece nu este o conversie prin lărgire. Versiunea modificată a programului de mai sus este prin urmare incorectă:

```
// *** Acest program nu se poate compila. ***

using System;

class DtoL {
  static void Main() {
    long L;
    double D;

    D = 100123285.0;
    L = D;  ←  Nu se poate face conversie automată din double în long

    Console.WriteLine("L si D: " + L + " " + D);
  }
}
```

În plus față de restricțiile prezentate, nu există conversii automate între **decimal** și **float** sau **double** și nici între tipurile numerice și **char** sau **bool**. Tipurile **char** și **bool** nu sunt, de asemenea, compatibile unul cu celălalt.

Conversia explicită a tipurilor incompatibile

Deși conversiile automate de tip sunt utile, ele nu rezolvă toate necesitățile de programare, deoarece acoperă numai conversiile prin lărgire între tipuri compatibile. Pentru toate celelalte cazuri, trebuie să utilizați un cast. Un *cast* este o directivă către compilator de a converti un anumit tip în altul. Acesta presupune deci o conversie explicită de tip. Forma generală a unui cast este:

(tip-țintă) expr;

Aici, *tip-țintă* reprezintă tipul la care dorim să convertim expresia specificată prin *expr*. De exemplu, doriți ca tipul expresiei **x/y** să fie **int**, puteți scrie:

```
double x, y;
//...
(int) (x / y);
```


Aici, chiar dacă **x** și **y** sunt de tipul **double**, castul convertește rezultatul expresiei la tipul **int**. Parantezele care înconjoară **x/y** sunt obligatorii. Altfel, castul la **int** s-ar fi aplicat numai lui **x** și nu rezultatului împărțirii. Castul a fost necesar aici, deoarece nu se efectuează conversie automată de la **double** la **int**.

Ori de câte ori un cast determină o *conversie prin restrângere*, informația poate fi pierdută. De exemplu, castul de la **long** la **int** va determina pierderea informației dacă valoarea **long** depășește domeniul tipului **int**, deoarece biții cei mai semnificativi sunt trunchiați. Atunci când o valoare în virgulă mobilă este convertită la un tip întreg, partea fracționară se pierde de asemenea prin trunchiere. De exemplu, dacă se atribuie valoarea 1.23 unui întreg, valoarea rezultată va fi 1. Componenta fracționară 0.23 se pierde.

Următorul program prezintă câteva conversii de tip care necesită casturi:

```
// Prezentarea casturilor

using System;

class CastDemo {
  static void Main() {
    double x, y;
    byte b;
    int i;
    char ch;

    x = 10.0;
    y = 3.0;

    i = (int) (x / y); // cast din double in int   ←  conversie cu trunchiere
    Console.WriteLine("Rezultatul intreg al lui x / y: " + i);

    i = 100;
    b = (byte) i;  ←  Nu se pierde informație. Valoarea 100 încape într-un byte.
    Console.WriteLine("Valoarea lui b: " + b);

    i = 257;
    b = (byte) i;  ←  Se pierde informație. Un byte nu poate memora valoarea 257.
    Console.WriteLine("Valoarea lui b: " + b);

    b = 88; // codul ASCII pentru X
    ch = (char) b;  ←  cast între tipuri incompatibile
    Console.WriteLine("ch: " + ch);
  }
}
```

Textul afișat de program este prezentat în continuare:

```
Rezultatul intreg al lui x / y: 3
Valoarea lui b: 100
Valoarea lui b: 1
ch: X
```

În program, castul expresiei (**x / y**) la **int** determină trunchierea părții fracționare și deci pierderea de informație. În continuare, nu se pierde informație atunci când **b**

primeşte valoarea 100, deoarece un **byte** poate memora valoarea 100. Când încercăm însă să-i atribuim lui **b** valoarea 257, are loc o pierdere de informaţie deoarece 257 depăşeşte domeniul de valori al tipului **byte**. În fine, nu se pierde informaţie, dar este necesar un cast atunci când atribuim o valoare de tip **byte** unei variabile **char**.

Exerciţii la minut

- Ce este un cast ?
- Poate fi o valoare **short** să fie atribuită unui **int** fără cast ? Se poate atribui un **byte** unui **char** fără cast ?
- Cum se poate rescrie următoarea instrucţiune ?

```
x = x + 23;
```

Precedenţa operatorilor

Tabelul 2-3 prezintă ordinea precedenţei pentru toţi operatorii din C#, de la prioritatea cea mai mare la cea mai scăzută. tabelul include câţiva operatori care vor fi prezentaţi ulterior în această carte.

Tabelul 2-3 Precedenţa operatorilor în C#

Prioritate maximă
`() [] . ++(postfix) --(postfix) checked new sizeof typeof unchecked`
`! ~ (cast) +(unar) ++(prefix) --(prefix)`
`* / %`
`+ -`
`<< >>`
`< > <= >= is`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= op=`
Prioritate minimă

- Un cast este o conversie explicită de tip.
- Da. Nu, deoarece sunt tipuri incompatibile.
- Instrucţiunea se poate rescrie astfel:

```
x += 23;
```


Expresii

Operatorii, literalii şi variabilele reprezintă elementele din care sunt alcătuite *expresiile*. În C#, o expresie este orice combinaţie validă a acestor elemente. Poate că deja cunoaşteţi forma generală a unei expresii dintr-un alt limbaj de programare, sau din algebră. Există însă câteva aspecte legate de expresii pe care le vom discuta în cele ce urmează.

Conversia tipurilor în expresii

În cadrul unei expresii, este posibil să amestecăm două sau mai multe tipuri de date, atât timp cât acestea sunt compatibile între ele. De exemplu, puteţi combina **short** şi **long** într-o expresie, deoarece acestea sunt ambele tipuri numerice. Dacă într-o expresie sunt implicate mai multe tipuri diferite, acestea sunt convertite la un tip comun, operaţie cu operaţie.

Conversiile se realizează utilizând *regulile de promovare a tipurilor* din C#. Iată algoritmul definit de aceste reguli pentru operaţiile binare:

DACĂ un operand este **decimal**, ATUNCI şi celălalt operand este promovat la **decimal** (cu excepţia cazului în care este de tipul **float** sau **double**, caz în care se produce o eroare).
ALTFEL DACĂ unul dintre operanzi este **double**, al doilea este promovat la **double**.
ALTFEL DACĂ unul dintre operanzi este **float**, al doilea este promovat la **float**.
ALTFEL DACĂ unul dintre operanzi este **ulong**, al doilea este promovat la **ulong** (numai dacă nu este de tip **sbyte**, **short**, **int** sau **long**, caz în care apare o eroare).
ALTFEL DACĂ un operand este **long**, al doilea este promovat la **long**.
ALTFEL DACĂ un operand este **uint**, iar al doilea este de tip **sbyte**, **short** sau **int**, ambii sunt promovaţi la **long**.
ALTFEL DACĂ un operand este **uint**, al doilea este promovat la **uint**.
ALTFEL ambii operanzi sunt promovaţi la **int**.

Există câteva aspecte care merită discutate referitoare la regulile de promovare. În primul rând, nu toate tipurile pot fi amestecate într-o expresie. Mai precis, nu există o conversie implicită de la **float** sau **double** la **decimal**, iar tipul **ulong** nu poate fi combinat cu nici un tip întreg cu semn. Combinarea acestor tipuri se poate face numai utilizând o conversie explicită (cast).

În al doilea rând, priviţi cu atenţie ultima regulă. Aceasta spune că dacă nici una din regulile anterioare nu se poate aplica, atunci toţi ceilalţi operanzi sunt promovaţi la **int**. În consecinţă, în cadrul expresiilor, toate valorile **char**, **sbyte**, **byte**, **ushort** sau **short** sunt promovate la **int** în scopul efectuării unor calcule. Această regulă este numită *promovarea întregilor*. Ea ne asigură de asemenea că rezultatul oricărei operaţii aritmetice nu poate avea un tip inferior lui **int**.

Este esențial să înțelegeți că promovările de tip se aplică numai asupra valorilor cu care se lucrează la evaluarea unei expresii. De exemplu, chiar dacă valoarea unei variabile **byte** este promovată la **int** în interiorul unei expresii, în afara expresiei aceasta rămâne de tip **byte**. Promovarea tipurilor afectează numai evaluarea expresiilor.

Promovarea tipurilor poate însă conduce și la rezultate oarecum neașteptate. De exemplu, atunci când o operație aritmetică implică două valori **byte**, are loc succesiunea: mai întâi, operanzii **byte** sunt promovați la **int**. Apoi se efectuează operația, care întoarce ca rezultat un **int**. Așadar, rezultatul unei operații în care au fost implicate două valori **byte** este un **int**. Aceasta nu concordă întru totul cu ce v-ați fi așteptat a priori. Să considerăm următorul program:

```
// Surprize legate de promovare !

using System;

class PromDemo {
  static void Main() {
    byte b;
    int i;

    b = 10;
    i = b * b;  // corect, nu trebuie cast

    b = 10;
    b = (byte) (b * b);  // cast obligatoriu !!!

    Console.WriteLine("i si b: " + i + " " + b);
  }
}
```

castul nu este necesar pentru că rezultatul este promovat la **int**

este nevoie de cast pentru a atribui un **int** unui **byte**!

Oarecum contrar așteptărilor, nu avem nevoie de cast când atribuim **b * b** lui **i**, deoarece **b** este promovat la **int** atunci când se evaluează expresia. Când însă încercați să atribuiți **b * b** lui **b**, aveți nevoie de cast – castul înapoi la **byte** ! Amintiți-vă de aceasta atunci când vă apar erori neașteptate de incompatibilitate de tip la expresii care aparent arată în regulă.

Aceeași situație se întâlnește și la efectuarea unor operații asupra datelor de tip **char**. De exemplu, în secvența următoare, castul înapoi la **char** este obligatoriu din cauza promovării variabilelor **ch1** și **ch2** la **int** în cadrul expresiei:

```
char ch1 = 'a', ch2 = 'b';
ch1 = (char) (ch1 + ch2);
```

Fără cast, rezultatul de tip **int** al adunării lui **ch1** și **ch2** nu putea fi atribuit unei variabile de tip **char**.

Castul nu se utilizează numai la conversia tipurilor în instrucțiunile de atribuire. De exemplu, să considerăm următorul program. Acesta utilizează un cast la **double** pentru a obține partea fracționară dintr-o împărțire întreagă.


```
// Utilizarea unui cast

using System;

class UseCast {
  static void Main() {
    int i;

    for(i = 0; i < 5; i++) {
      Console.WriteLine(i + " / 3: " + i / 3);
      Console.WriteLine(i + " / 3 cu zecimale: {0:#.##}",
                        (double) i / 3);
      Console.WriteLine();
    }
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
0 / 3: 0
0 / 3 cu zecimale:

1 / 3: 0
1 / 3 cu zecimale: .33

2 / 3: 0
2 / 3 cu zecimale: .67

3 / 3: 1
3 / 3 cu zecimale: 1

4 / 3: 1
4 / 3 cu zecimale: 1.33
```

Sfatul expertului

Întrebare: Au loc promovări de tip și pentru operatorii unari (cu un singur operand – n.t.), cum ar fi minusul (-) unar ?

Răspuns: Da. Pentru operațiile unare, operanzii mai mici decât **int** (**byte**, **sbyte**, **short** și **ushort**) sunt promovați la **int**. De asemenea, operanzii de tipul **char** sunt promovați la **int**. Mai mult, dacă o valoare **uint** este negată, aceasta este promovată la **long**.

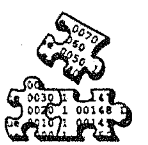
Spațierea și utilizarea parantezelor

Orice expresie în C# poate conține spații și caractere tab pentru a fi mai ușor de citit. De exemplu, următoarele două expresii sunt identice, a doua fiind însă mai ușor de citit:

```
x=10/y*(127/x);
x = 10 / y * (127/x);
```

Parantezele determină creşterea precedenţei operaţiilor pe care le conţin, la fel ca în algebră. Utilizarea parantezelor suplimentare sau redundante nu determină apariţia unor erori şi nici nu încetineşte evaluarea unei expresii. Vă încurajăm pe această cale să utilizaţi paranteze pentru a stabili precis ordinea exactă a evaluării, atât pentru dumneavoastră cât şi pentru cei care vor citi programul în viitor. De exemplu, care dintre următoarele expresii este mai uşor de citit ?

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```



Proiectul 2-3: Calculul ratelor lunare pentru un împrumut

După cum am precizat ceva mai devreme, tipul **decimal** este în general adecvat pentru calcule financiare. În acest proiect, veţi utiliza tipul **decimal** într-un caz real. Veţi crea un program care calculează ratele lunare ale unui împrumut, contractat de exemplu pentru cumpărarea unei maşini. Cunoscând valoarea împrumutului, numărul de rate pe an şi rata dobânzii, programul va calcula valoarea ratei lunare. Deoarece acest calcul este de natură financiară, este indicat să utilizăm tipul **decimal** pentru reprezentarea datelor. Proiectul prezintă de asemenea conversia de tipuri şi o altă metodă de bibliotecă din C#: **Pow()**.

Pentru calculul ratelor, veţi utiliza următoarea formulă:

$$\text{Rata} = \frac{\text{RataDob} \cdot \dfrac{\text{SumaTot}}{\text{RatePeAn}}}{1 - \left(\dfrac{\text{RataDob}}{\text{RatePeAn}} + 1\right)^{-\text{RatePeAn} \cdot \text{NrAni}}}$$

unde RataDob reprezintă rata dobânzii, SumaTot este valoarea împrumutată, RatePeAn este numărul de plăţi care se efectuează într-un an, iar NrAni specifică durata în ani a împrumutului.

Observaţi că la numitorul formulei, trebuie să ridicaţi o valoare la o anumită putere. Pentru aceasta, veţi utiliza o altă metodă matematică din C#: **Math.Pow()**. Modul în care trebuie să o apelaţi este prezentat mai jos:

rezultat = **Math.Pow**(*baza, exp);*

Pow() întoarce valoarea bazei *baza*, ridicată la puterea *exp*. Parametrii lui **Pow()** trebuie să fie de tipul **double**; metoda întoarce tot o valoare de tip **double**. Rezultă că este nevoie să utilizaţi un cast pentru conversia de la **double** la **decimal**.

Pas cu pas

1. Creaţi un fişier nou, numit **RegPay.cs**.
2. Iată care sunt variabilele pe care le veţi utiliza în program:

```
decimal SumaTot;          // suma imprumutata
decimal RataDob;          // rata dobanzii ca valoare zecimala, de exemplu 0.075
decimal RatePeAn;         // numarul de rate pe an
```



```
decimal NrAni;                   // numarul anilor
decimal Rata;                    // valoarea ratei lunare
decimal numarator, numitor;      // variabile temporare de lucru
double b, e;     // baza si exponentul la apelul lui Pow()
```

Deoarece majoritatea calculelor se vor efectua utilizând tipul de date **decimal**, majoritatea variabilelor vor fi de tipul **decimal**.

Remarcaţi că fiecare declaraţie de variabilă este însoţită de un comentariu care explică modul de utilizare a variabilei. Acestea vor ajuta pe oricine vă va citi programul să înţeleagă rolul fiecărei variabile. Deşi pentru majoritatea programelor scurte din această carte nu vom include comentarii atât de amănunţite, aceasta este o regulă pe care este bine să o respectaţi pe măsură ce programele dumneavoastră devin mai lungi şi mai complexe.

3. Adăugaţi următoarele linii de cod, care precizează datele unui împrumut real. În acest caz particular, suma totală este de 10 000 USD, rata dobânzii de 7.5 procente, numărul ratelor anuale de 12, iar durata împrumutului de 5 ani.

```
SumaTot = 10000.00m;
RataDob = 0.075m;
RatePeAn = 12.0m;
NrAni = 5.0m;
```

4. Adăugaţi acum liniile care realizează calculul financiar propriu-zis:

```
numarator = RataDob * SumaTot / RatePeAn;

e = (double) -(RatePeAn * NrAni);

b = (double) (RataDob / RatePeAn) + 1;

numitor = 1 - (decimal) Math.Pow(b, e);

Rata = numarator / numitor;
```

Remarcaţi că trebuie să utilizaţi casturi pentru a construi parametrii care sunt transmişi lui **Pow()** şi apoi pentru a converti valoarea întoarsă. Reamintiţi-vă că în C# nu se efectuează conversie automată între tipurile **decimal** şi **double**.

5. Încheiaţi programul cu afişarea valorii ratei lunare, după cum puteţi observa mai jos:

```
Console.WriteLine("Rata este {0:C}", Rata);
```

6. Iată listingul complet al programului **RegPay.cs**:

```
/*
   Proiectul 2-3

   Calculul ratei lunare pentru un imprumut

   Denumiti fisierul RegPay.cs
*/

using System;

class RegPay {
  static void Main() {
    decimal SumaTot;           // suma imprumutata
    decimal RataDob;           // rata dobanzii ca valoare zecimala,
                               // de exemplu 0.075
    decimal RatePeAn;          // numarul de rate pe an
```

```
    decimal NrAni;              // numarul anilor
    decimal Rata;               // valoarea ratei lunare
    decimal numarator, numitor; // variabile temporare de lucru
    double b, e;                // baza si exponentul la apelul lui Pow()

    SumaTot = 10000.00m;
    RataDob = 0.075m;
    RatePeAn = 12.0m;
    NrAni = 5.0m;

    numarator = RataDob * SumaTot / RatePeAn;

    e = (double) -(RatePeAn * NrAni);
    b = (double) (RataDob / RatePeAn) + 1;

    numitor = 1 - (decimal) Math.Pow(b, e);
    Rata = numarator / numitor;

    Console.WriteLine("Rata este {0:C}", Rata);

  }
}
```

Programul va afişa următorul rezultat:

```
Rata este $200.38
```

Înainte de a trece mai departe, puteţi încerca să utilizaţi programul la calculul ratelor lunare modificând suma împrumutată, durata sau rata dobânzii.

Verificarea cunoştinţelor

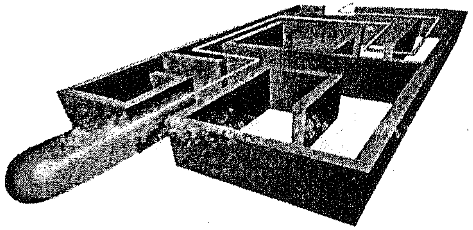
1. De ce în C# domeniul şi comportamentul tipurilor simple sunt precizate strict?
2. Care este tipul caracter în C#, şi care este diferenţa dintre acest tip şi tipul caracter utilizat în alte limbaje de programare ?
3. O variabilă **bool** poate lua orice valoare, deoarece orice valoare nenulă reprezintă valoarea logică „adevărat". Este adevărat sau fals ?
4. Se dă textul afişat de un program:
```
One
Two
Three
```
   Utilizând un singur şir de caractere şi secvenţe escape, scrieţi instrucţiunea **WriteLine()** care a afişat acest text.
5. Unde este greşeala în secvenţa de mai jos ?
```
for(i = 0; i < 10; i++) {
   int sum;

   sum = sum + i;
}
Console.WriteLine("Suma este: " + sum);
```


6. Precizaţi diferenţa dintre forma prefixată şi cea postfixată ale operatorului de incrementare.
7. Arătaţi cum se poate utiliza un operator ŞI scurtcircuitat pentru a evita o eroare de împărţire la zero.
8. În cadrul unei expresii, la ce tip sunt promovate **byte** şi **short** ?
9. Care dintre tipurile de mai jos nu poate fi combinat într-o expresie cu o valoare de tip **decimal** ?
   A. **float**
   B. **int**
   C. **uint**
   D. **byte**
10. În general, când este necesar un cast ?
11. Scrieţi un program care determină toate numerele prime cuprinse între 1 şi 100.
12. Rescrieţi independent programul cu tabele de adevăr din proiectul 2-2 astfel încât să utilizeze literali „copie la indigo" cu caractere tab şi „linie nouă" incluse în locul secvenţelor escape.

Modulul 3

Instrucțiuni de control



Scopurile acestui modul

- **Introducerea datelor de la tastatură**
- **Aprofundarea instrucțiunilor `if` și `for`**
- **Studiul instrucțiunii `switch`**
- **Înțelegerea instrucțiunii `while`**
- **Utilizarea buclei `do-while`**
- **Întrebuințarea instrucțiunii `break`**
- **Utilizarea instrucțiunii `continue`**
- **Examinarea instrucțiunii `goto`**

În acest capitol, veți învăța instrucțiunile care controlează fluxul execuției unui program. Există trei categorii de astfel de instrucțiuni: instrucțiuni de *selecție*, care sunt **if** și **switch**; instrucțiuni de *iterare*, care sunt buclele **for**, **while**, **do-while** și **foreach** și, în fine, instrucțiuni de *salt*, care sunt **break**, **continue**, **goto** și **return**. Cu excepția instrucțiunilor **foreach** și **return**, care vor fi prezentate ulterior, celelalte instrucțiuni de control, printre care și instrucțiunile **if** și **for** care au fost deja prezentate pe scurt, sunt studiate în detaliu în acest modul.

Introducerea caracterelor de la tastatură

Înainte de a studia instrucțiunile de control din C#, vom face o scurtă paranteză care vă va permite să începeți să scrieți programe interactive. Exemplele prezentate până acum în această carte afișează o informație pe care utilizatorul o poate vizualiza, dar nu primesc informație *de la* utilizator. Așadar, ați utilizat până acum afișarea la consolă, dar nu și introducerea datelor de la consolă (de fapt, de la tastatură). Veți începe acum să utilizați intrarea în sistem citind caracterele introduse de la tastatură.

Pentru citirea unui caracter introdus de la tastatură, se apelează metoda **Console.Read()**. Aceasta așteaptă până când utilizatorul apasă o tastă și apoi întoarce un rezultat. Caracterul tastat este întors ca valoare întreagă, deci trebuie convertit explicit la **char** pentru a fi atribuit unei variabile de tipul **char**. În mod implicit, datele introduse de la consolă sunt direcționate într-un *buffer de linie*; prin urmare, caracterele pe care le-ați tastat sunt trimise către programul dumneavoastră numai după ce apăsați ENTER. Iată un program care citește un caracter de la tastatură:


```
// Citirea unui caracter de la tastatura

using System;

class KbIn {
  static void Main() {
    char ch;

    Console.Write("Apasati o tasta si apoi ENTER: ");

    ch = (char) Console.Read();   // citeste un caracter

    Console.WriteLine("Ati apasat tasta: " + ch);
  }
}
```

Citește un caracter de la tastatură.

În continuare, prezentăm dialogul care apare la o rulare a programului:

```
Apasati o tasta si apoi ENTER: t
Ati apasat tasta: t
```

Faptul că metoda **Read()** citește de fapt dintr-un buffer de linie poate cauza anumite neplăceri. La apăsarea tastei ENTER, în stream-ul de intrare se introduce secvența de caractere „revenire la cap de rând" (carriage-return) și „linie nouă" (line-feed). Mai mult, aceste caractere așteaptă în buffer-ul de intrare până când sunt citite. În unele aplicații, trebuie deci să le ștergeți din buffer (adică să le citiți) înainte de a trece la următoarea operație de introducere de date. Vom prezenta un exemplu în acest sens pe parcursul acestui modul.

Exerciții la minut

- Cum puteți citi un caracter de la tastatură ?
- Ce înțelegeți prin direcționarea într-un buffer de linie ?

Instrucțiunea `if`

Instrucțiunea **if** a fost prezentată în modulul 1. În acest modul, o vom studia în detaliu. Forma completă a instrucțiunii **if** este:

if(*condiție*) *instrucțiune*;
else *instrucțiune*;

țintele pentru **if** și **else** fiind instrucțiuni simple. Clauza **else** este opțională. Atât ținta lui **if**, cât și cea pentru **else** pot fi însă blocuri de instrucțiuni. Forma generală a instrucțiunii **if** care utilizează blocuri de instrucțiuni este:

- Pentru citirea unui caracter, apelați **Console.Read()**.
- Atunci când intrarea este direcționată într-un buffer de linie, trebuie să apăsați ENTER pentru a trimite ceea ce ați tastat către programul dumneavoastră.

```
if(condiție)
{
  secvență de instrucțiuni
}
else
{
  secvență de instrucțiuni
}
```

Dacă expresia condițională este adevărată, atunci se execută ținta lui **if**; altfel, dacă există, se execută ținta lui **else**. În nici un caz nu este posibil ca ambele instrucțiuni să fie executate. Expresia condițională care controlează instrucțiunea **if** trebuie să producă un rezultat de tipul **bool**.

Pentru a prezenta instrucțiunea **if**, vom dezvolta un joc simplu al cărui scop este ghicirea unei litere, potrivit pentru copiii de vârstă mică. Într-o primă versiune a jocului, programul cere jucătorului să ghicească o literă de la A la Z. Dacă jucătorul ghicește litera, programul răspunde afișând mesajul ** **Corect** **. Programul este prezentat în continuare:

```
// Jocul "Ghiceste litera"

using System;

class Guess {
  public static void Main() {
    char ch, answer = 'K';

    Console.WriteLine("Ma gandesc la o litera de la A la Z.");
    Console.Write("O poti ghici: ");

    ch = (char) Console.Read();   // citeste un caracter de la tastatura

    if(ch == answer) Console.WriteLine("** Corect **");
  }
}
```

Programul îl informează pe jucător, iar apoi citește un caracter de la tastatură. Utilizând o instrucțiune **if**, programul verifică dacă acest caracter coincide cu răspunsul stabilit, care este K în acest caz. Dacă se tastează K, programul afișează mesajul. Atunci când încercați acest program, rețineți că litera K introdusă trebuie să fie majusculă.

Dezvoltând jocul, următoarea versiune utilizează clauza **else** pentru a afișa un alt mesaj atunci când litera tastată nu coincide cu răspunsul.

```
// "Ghiceste litera", versiunea a doua

using System;

class Guess2 {
  public static void Main() {
    char ch, answer = 'K';
```



```
    Console.WriteLine("Ma gandesc la o litera de la A la Z.");
    Console.Write("O poti ghici: ");

    ch = (char) Console.Read();   // citeste un caracter de la tastatura

    if(ch == answer) Console.WriteLine("** Corect **");
    else Console.WriteLine("...Imi pare rau, ai gresit.");
  }
}
```

If-uri imbricate

Un **if** *imbricat* este o instrucțiune **if** care este ținta unui alt **if** sau a unui **else**. Utilizarea **if**-urilor imbricate este o practică foarte frecventă în programare. Aspectul cel mai important de reținut privitor la **if**-urile imbricate este că orice clauză **else** este perechea celui mai apropiat **if** din același bloc cu ea și care nu a fost deja asociat cu o altă clauză **else**. Iată în continuare un exemplu:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d;
  else a = c; // acest else este pereche cu if(k > 100)
}
else a = d; // acest else este pereche cu if(I == 100)
```

Acest **if** corespunde cu acest **else**.

Acest **if** corespunde cu acest **else**.

După cum arată și comentariile, clauza **else** finală nu este asociată cu **if(j < 20)**, deoarece acesta nu aparține aceluiași bloc (chiar dacă este cel mai apropiat **if** fără **else**). Clauza **else** finală se asociază cu **if(i == 10)**. Clauza **else** internă este pereche cu **if(k > 100)**, acesta fiind **if**-ul cel mai apropiat din același bloc.

Puteți utiliza **if**-urile imbricate pentru încă o îmbunătățire a jocului „ghicește litera". Acum, jucătorul va primi în plus o informație în cazul unei alegeri incorecte.

```
// "Ghiceste litera", versiunea a treia

using System;

class Guess3 {
  public static void Main() {
    char ch, answer = 'K';

    Console.WriteLine("Ma gandesc la o litera de la A la Z.");
    Console.Write("O poti ghici: ");

    ch = (char) Console.Read();   // citeste un caracter de la tastatura

    if(ch == answer) Console.WriteLine("** Corect **");
    else {
      Console.Write("...Imi pare rau, litera e ");

      // un if imbricat
      if(ch < answer) Console.WriteLine("prea mica.");
```

Un **if** imbricat

```
        else Console.WriteLine("prea mare.");
      }
    }
}
```

Un exemplu de rulare a programului este prezentat mai jos:

```
Ma gandesc la o litera de la A la Z.
O poti ghici: Z
...Imi pare rau, litera e prea mare.
```

Scara if-else-if

O construcție uzuală în programare bazată pe `if`-urile imbricate este scara *if-else-if*. Aceasta are forma:

```
if(condiție)
  instrucțiune;
else if(condiție)
  instrucțiune;
else if(condiție)
  instrucțiune;
.
.
.
else
  instrucțiune;
```

Expresiile condiționale sunt evaluate de sus în jos. De îndată ce se întâlnește o condiție adevărată, se execută instrucțiunea asociată cu aceasta, restul scării fiind ignorat. Dacă nici una dintre condiții nu este adevărată, se execută instrucțiunea `else` finală. Clauza `else` finală se comportă în multe cazuri ca o condiție implicită; cu alte cuvinte, dacă toate celelalte teste eşuează, ultima instrucțiune `else` finală este cea care se execută. Dacă nu există o clauză `else` finală şi toate celelalte condiții sunt false, nu se va executa nimic.

Programul următor prezintă utilizarea scării `if-else-if`:

```
// Prezinta scara if-else-if.

using System;

class Ladder {
  public static void Main() {
    int x;

    for(x = 0; x < 6; x++) {
      if(x == 1)
        Console.WriteLine("x este unu");
      else if(x == 2)
        Console.WriteLine("x este doi");
      else if(x == 3)
        Console.WriteLine("x este trei");
```



```
      else if(x == 4)
        Console.WriteLine("x este patru");
      else
        Console.WriteLine("x nu este intre 1 si 4"); // <- Aceasta este instrucțiunea implicită
    }
  }
}
```

Programul va afişa următorul text:

```
x nu este intre 1 si 4
x este unu
x este doi
x este trei
x este patru
x nu este intre 1 si 4
```

După cum puteți observa, clauza `else` implicită se execută numai dacă toate instrucțiunile `if` precedente eşuează.

Exerciții la minut

- Ce tip trebuie să aibă condiția care controlează instrucțiunea `if` ?
- Cu care `if` se asociază o clauză `else` ?
- Ce este o scară `if-else-if` ?

Instrucțiunea switch

Cea de-a doua instrucțiune de selecție din C# este `switch`. Instrucțiunea `switch` implementează o ramificație cu mai multe căi. Astfel, ea permite unui program să selecteze una din mai multe variante. Chiar dacă şi o secvență de `if`-uri imbricate poate implementa teste cu ramificații multiple, în majoritatea situațiilor instrucțiunea `switch` reprezintă o abordare mai eficientă. Instrucțiunea funcționează astfel: se verifică succesiv dacă valoarea unei expresii este egală cu fiecare din constantele dintr-o listă. Atunci când se detectează potrivirea cu un element al listei, se execută secvența de instrucțiuni asociată cu valoarea respectivă. Forma generală a instrucțiunii `switch` este:

```
switch(expresie) {
  case constanta1:
    secvență de instrucțiuni
    break;
```

- Condiția care controlează un `if` trebuie să fie de tipul `bool`.
- Un `else` se asociază întotdeauna cu cel mai apropiat `if` din blocul său care nu are deja asociat un `else`.
- O scară `if-else-if` este o secvență de instrucțiuni `if-else` imbricate.

```
    case constanta2:
        secvență de instrucțiuni
        break;
    case constanta3:
        secvență de instrucțiuni
        break;
    .
    .
    .
    default:
        secvență de instrucțiuni;
        break;
}
```

Expresia care controlează instrucțiunea **switch** trebuie să fie de unul din tipurile întregi, cum ar fi **char**, **byte**, **short** sau **int**, sau de tipul **string** (care va fi prezentat ceva mai târziu în lucrare). În consecință, expresiile în virgulă mobilă, de exemplu, nu sunt permise. În mod uzual, expresia care controlează instrucțiunea **switch** este pur și simplu o variabilă. Constantele **case** trebuie să fie literali de un tip compatibil cu expresia. Nu este posibil ca două constante **case** din același **switch** să aibă valori identice.

Secvența de instrucțiuni **default** se execută dacă nici una din constantele **case** nu corespunde cu expresia. Prezența clauzei **default** este opțională; dacă aceasta nu există, nu se execută nimic în cazul în care toate testele eșuează. La detectarea unei potriviri, se execută instrucțiunile asociate clauzei **case** corespunzătoare până cînd se întâlnește o instrucțiune **break**.

Programul următor utilizează instrucțiunea **switch**:

```
// Prezentarea lui switch.

using System;

class SwitchDemo {
  public static void Main() {
    int i;

    for(i = 0; i < 10; i++)
      switch(i) {
        case 0:
          Console.WriteLine("i este zero");
          break;
        case 1:
          Console.WriteLine("i este unu");
          break;
        case 2:
          Console.WriteLine("i este doi");
          break;
        case 3:
          Console.WriteLine("i este trei");
          break;
        case 4:
          Console.WriteLine("i este patru");
          break;
        default:
          Console.WriteLine("i este mai mare sau egal cu cinci");
          break;
      }
  }
}
```


Programul afișează textul prezentat mai jos:

```
i este zero
i este unu
i este doi
i este trei
i este patru
i este mai mare sau egal cu cinci
i este mai mare sau egal cu cinci
i este mai mare sau egal cu cinci
i este mai mare sau egal cu cinci
i este mai mare sau egal cu cinci
```

După cum se poate observa, la fiecare iterație a buclei, se execută instrucțiunile asociate constantei **case** care este egală cu **i**. Toate celelalte instrucțiuni **case** sunt ignorate.Când **i** devine mai mare sau egal cu 5, nici una din constantele **case** nu se mai potrivește și se execută instrucțiunea **default**.

În exemplul anterior, instrucțiunea **switch** a fost controlată de o variabilă **int**. După cum am precizat, puteți controla un **switch** cu orice tip întreg, inclusiv **char**. Iată un exemplu care utilizează o expresie **char** și constante **case** de tipul **char**:

```
// Utilizarea unei expresii char la controlul unui switch.

using System;

class SwitchDemo2 {
  public static void Main() {
    char ch;

    for(ch = 'A'; ch <= 'E'; ch++)
      switch(ch) {
        case 'A':
          Console.WriteLine("ch este A");
          break;
        case 'B':
          Console.WriteLine("ch este B");
          break;
        case 'C':
          Console.WriteLine("ch este C");
          break;
        case 'D':
          Console.WriteLine("ch este D");
          break;
        case 'E':
          Console.WriteLine("ch este E");
          break;
      }
  }
}
```

Textul afişat de acest program este prezentat mai jos:

```
ch este A
ch este B
ch este C
ch este D
ch este E
```

Observaţi că acest exemplu nu conţine o instrucţiune **default**. Reţineţi că utilizarea clauzei **default** este opţională. Atunci când nu este necesară, ea poate fi omisă.

În C#, se produce o eroare dacă secvenţa de instrucţiuni asociată cu un **case** continuă cu următorul **case**. Aceasta se numeşte *regula de interzicere a execuţiei în secvenţă*. Din acest motiv, toate secvenţele **case** se termină cu instrucţiunea **break**. (Este posibil să interziceţi continuarea execuţiei şi în alte moduri, dar instrucţiunea **break** este de departe metoda cel mai des utilizată.) Atunci când este întâlnită în secvenţa de instrucţiuni a unui **case**, **break** determină fluxul execuţiei programului să părăsească întreaga instrucţiune **switch**, continuând cu următoarea instrucţiune exterioară lui **switch**. Un alt aspect de reţinut este că nici clauza **default** nu trebuie să permită continuarea execuţiei, motiv pentru care de obicei şi ea se termină cu **break**.

Chiar dacă nu este permis ca o secvenţă **case** să se continue cu o alta, este posibil ca două sau mai multe instrucţiuni **case** să refere aceeaşi secvenţă de cod, cum se observă şi în acest exemplu:

```
switch(i) {
  case 1:
  case 2:
  case 3:  Console.WriteLine(" i este 1, 2 sau 3");
    break;
  case 4:  Console.WriteLine("i este 4");
    break;
}
```

Aceste **case**-uri referă aceeaşi secvenţă de cod.

În acest fragment, dacă **i** are valoarea 1, 2 sau 3, se execută prima instrucţiune **WriteLine()**. Dacă **i** este 4, atunci se execută a doua instrucţiune **WriteLine()**. Gruparea instrucţiunilor **case** nu încalcă regula interzicerii execuţiei în secvenţă, deoarece toate instrucţiunile grupate partajează aceeaşi secvenţă de cod.

Gruparea este o metodă frecvent utilizată atunci când mai multe instrucţiuni **case** pot partaja acelaşi cod. În exemplul următor, gruparea este utilizată pentru separarea literelor alfabetului în vocale şi consoane:

```
// Separarea literelor in vocale si consoane

using System;

class VocaleSiConsoane {
  public static void Main() {
    char ch;

    Console.Write("Introduceti o litera ");
    ch = (char) Console.Read();
```



```
    switch(ch) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
      case 'y':
        Console.WriteLine("Litera este o vocala");
        break;
      default:
        Console.WriteLine("Litera este o consoana");
        break;
    }
  }
}
```

Dacă am fi scris acest exemplu fără să grupăm instrucţiunile **case**, aceeaşi instrucţiune **WriteLine()** ar fi fost duplicată de şase ori. Gruparea clauzelor **case** evită această duplicare redundantă.

Sfatul expertului

Întrebare: În ce caz este preferabil să utilizăm o scară if-else-if în locul instrucţiunii switch atunci când implementăm o ramificaţie multiplă?

Răspuns: În general, veţi utiliza o scară **if-else-if** atunci când condiţiile care controlează selecţia nu se bazează pe o singură valoare. De exemplu, să considerăm următoarea secvenţă **if-else-if**:

```
if(x < 10)  //...
else if(y != 0)  //...
else if(! done) //...
```

Această secvenţă nu poate fi transcrisă într-o instrucţiune **switch** deoarece toate cele trei condiţii implică variabile diferite şi chiar de tipuri diferite. Care dintre acestea ar putea controla un **switch** ? De asemenea, este necesar să utilizaţi o scară **if-else-if** la testarea valorilor în virgulă mobilă, sau la testarea altor obiecte care nu sunt de unul din tipurile permise într-o expresie **switch**.

Instrucţiuni switch imbricate

Este posibil ca o instrucţiune **switch** să constituie o parte a unei secvenţe de instrucţiuni dintr-un **switch** exterior. Instrucţiunea interioară se numeşte **switch** imbricat. Constantele **case** din **switch**-ul interior şi cel exterior pot conţine valori comune, fără ca aceasta să genereze conflicte. De exemplu, următoarea secvenţă de cod este perfect corectă:

```
switch(ch1) {
  case 'A': Console.WriteLine("Acest A face parte din switch-ul exterior");
    switch(ch2) {
      case 'A':
        Console.WriteLine("Acest A face parte din switch-ul interior");
        break;
      case 'B': //...
    } // sfarsitul switch-ului interior
    break;
  case 'B': //...
```

switch imbricat

Sfatul expertului

Întrebare: În C, C++ şi Java, o secvenţă `case` poate continua cu următoarea. De ce nu se permite acest lucru şi în C# ?

Răspuns: Există două motive pentru care limbajul C# a instituit regula de interzicere a execuţiei în secvenţă pentru instrucţiunile **case**. În primul rând, compilatorul poate reordona în mod liber instrucţiunile **case**, în scopul optimizării codului. O astfel de rearanjare nu este posibilă dacă o secvenţă **case** se continuă cu următoarea. În al doilea rând, obligaţia ca fiecare **case** să se termine în mod explicit evită ca din greşeală, programatorul să permită continuarea unei secvenţe **case** cu următoarea.

Exerciţii la minut

- Ce tip trebuie să aibă expresia care controlează un **switch** ?
- Ce se întâmplă atunci când expresia care controlează un **switch** se potriveşte cu una din constantele **case** ?
- Este posibil ca o secvenţă de instrucţiuni asociată unei constante **case** să se continue cu următorul **case** ?

Proiectul 3-1: Construirea unui sistem de help pentru C#

În acest proiect, se va construi un sistem simplu de help, care afişează sintaxa instrucţiunilor de control din C#. Programul afişează un meniu care conţine instrucţiunile de control şi aşteaptă ca să alegeţi una dintre acestea. După alegerea unei instrucţiuni, programul afişează sintaxa acesteia. În această primă versiune a programului, se vor documenta numai instrucţiunile **if** şi **switch**. Celelalte instrucţiuni de control se vor adăuga în proiecte ulterioare.

- Expresia asociată unui **switch** trebuie să fie de un tip întreg sau de tipul **string**.
- Atunci când se detectează potrivirea cu una din constantele **case**, se execută secvenţa de instrucţiuni asociată acelei constante.
- Nu, deoarece în C# se aplică regula interzicerii execuţiei în secvenţă pentru instrucţiunile **switch**.

Pas cu pas

1. Creaţi un fişier cu numele **Help.cs**.
2. Programul începe cu afişarea meniului următor:

```
Help la:
  1. if
  2. switch
Alegerea dumneavoastra:
```

Pentru aceasta, se utilizează secvenţa de instrucţiuni prezentată mai jos:

```
Console.WriteLine("Help la:");
Console.WriteLine("  1. if");
Console.WriteLine("  2. switch");
Console.WriteLine("Alegerea dumneavoastră: ");
```

3. Programul determină alegerea utilizatorului apelând **Console.Read()**, ca mai jos:

```
choice = (char) Console.Read();
```

4. După determinarea selecţiei, programul utilizează instrucţiunea **switch** prezentată mai jos pentru afişarea sintaxei instrucţiunii selectate:

```
switch(choice) {
  case '1':
    Console.WriteLine("Instructiunea if:\n");
    Console.WriteLine("if(conditie) instructiune;");
    Console.WriteLine("else instructiune;");
    break;
  case '2':
    Console.WriteLine("Instructiunea switch:\n");
    Console.WriteLine("switch(expresie) {");
    Console.WriteLine("  case constanta:");
    Console.WriteLine("  secventa de instructiuni");
    Console.WriteLine("  break;");
    Console.WriteLine("  // ...");
    Console.WriteLine("}");
    break;
  default:
    Console.WriteLine("Selectie incorecta.");
    break;
}
```

Observaţi cum clauza **default** captează selecţiile incorecte. De exemplu, dacă utilizatorul introduce 3, nici o constantă **case** nu se potriveşte, ceea ce va duce la executarea secvenţei **default**.

5. Iată listingul complet al programului **Help.cs**:

```
/*
  Proiectul 3-1

  Un sistem simplu de help
*/

using System;
```

```
switch(ch1) {
  case 'A': Console.WriteLine("Acest A face parte din switch-ul exterior");
    switch(ch2) {
     case 'A':
       Console.WriteLine("Acest A face parte din switch-ul interior");
       break;
     case 'B': //...
    } // sfarsitul switch-ului interior
   break;
  case 'B': //...
```

switch imbricat

Sfatul expertului

Întrebare: În C, C++ şi Java, o secvenţă case poate continua cu următoarea. De ce nu se permite acest lucru şi în C# ?

Răspuns: Există două motive pentru care limbajul C# a instituit regula de interzicere a execuţiei în secvenţă pentru instrucţiunile **case**. În primul rând, compilatorul poate reordona în mod liber instrucţiunile **case**, în scopul optimizării codului. O astfel de rearanjare nu este posibilă dacă o secvenţă **case** se continuă cu următoarea. În al doilea rând, obligaţia ca fiecare **case** să se termine în mod explicit evită ca din greşeală, programatorul să permită continuarea unei secvenţe **case** cu următoarea.

Exerciţii la minut

- Ce tip trebuie să aibă expresia care controlează un **switch** ?
- Ce se întâmplă atunci când expresia care controlează un **switch** se potriveşte cu una din constantele **case** ?
- Este posibil ca o secvenţă de instrucţiuni asociată unei constante **case** să se continue cu următorul **case** ?

Proiectul 3-1: Construirea unui sistem de help pentru C#

În acest proiect, se va construi un sistem simplu de help, care afişează sintaxa instrucţiunilor de control din C#. Programul afişează un meniu care conţine instrucţiunile de control şi aşteaptă ca să alegeţi una dintre acestea. După alegerea unei instrucţiuni, programul afişează sintaxa acesteia. În această primă versiune a programului, se vor documenta numai instrucţiunile **if** şi **switch**. Celelalte instrucţiuni de control se vor adăuga în proiecte ulterioare.

- Expresia asociată unui **switch** trebuie să fie de un tip întreg sau de tipul **string**.
- Atunci când se detectează potrivirea cu una din constantele **case**, se execută secvenţa de instrucţiuni asociată acelei constante.
- Nu, deoarece în C# se aplică regula interzicerii execuţiei în secvenţă pentru instrucţiunile **switch**.

Pas cu pas

1. Creaţi un fişier cu numele **Help.cs**.
2. Programul începe cu afişarea meniului următor:

```
Help la:
  1. if
  2. switch
Alegerea dumneavoastra:
```

Pentru aceasta, se utilizează secvenţa de instrucţiuni prezentată mai jos:

```
Console.WriteLine("Help la:");
Console.WriteLine("  1. if");
Console.WriteLine("  2. switch");
Console.WriteLine("Alegerea dumneavoastră: ");
```

3. Programul determină alegerea utilizatorului apelând **Console.Read()**, ca mai jos:

```
choice = (char) Console.Read();
```

4. După determinarea selecţiei, programul utilizează instrucţiunea **switch** prezentată mai jos pentru afişarea sintaxei instrucţiunii selectate:

```
switch(choice) {
  case '1':
    Console.WriteLine("Instructiunea if:\n");
    Console.WriteLine("if(conditie) instructiune;");
    Console.WriteLine("else instructiune;");
    break;
  case '2':
    Console.WriteLine("Instructiunea switch:\n");
    Console.WriteLine("switch(expresie) {");
    Console.WriteLine("  case constanta:");
    Console.WriteLine("  secventa de instructiuni");
    Console.WriteLine("  break;");
    Console.WriteLine("  // ...");
    Console.WriteLine("}");
    break;
  default:
    Console.WriteLine("Selectie incorecta.");
    break;
}
```

Observaţi cum clauza **default** captează selecţiile incorecte. De exemplu, dacă utilizatorul introduce 3, nici o constantă **case** nu se potriveşte, ceea ce va duce la executarea secvenţei **default**.

5. Iată listingul complet al programului **Help.cs**:

```
/*
   Proiectul 3-1

   Un sistem simplu de help
*/

using System;
```

```
class Help {
  public static void Main() {
    char choice;

    Console.WriteLine("Help la:");
    Console.WriteLine("  1. if");
    Console.WriteLine("  2. switch");
    Console.WriteLine("Alegerea dumneavoastra: ");
    choice = (char) Console.Read();

    Console.WriteLine("\n");

    switch(choice) {
      case '1':
        Console.WriteLine("Instructiunea if:\n");
        Console.WriteLine("if(conditie) instructiune;");
        Console.WriteLine("else instructiune;");
        break;
      case '2':
        Console.WriteLine("Instructiunea switch:\n");
        Console.WriteLine("switch(expresie) {");
        Console.WriteLine("  case constanta:");
        Console.WriteLine("secventa de instructiuni");
        Console.WriteLine(" break;");
        Console.WriteLine("  // ...");
        Console.WriteLine("}");
        break;
      default:
        Console.WriteLine("Selectie incorecta.");
        break;
    }
  }
}
```

Iată ce afişează programul la rulare de probă:

```
Help la:
  1. if
  2. switch
Alegerea dumneavoastra: 1

Instructiunea if:

if(conditie) instructiune;
else instructiune;
```

Bucla for

Am utilizat o formă simplă a buclei **for** încă din modulul 1. Este posibil să nu intuiţi încă adevărata putere şi flexibilitate a buclei **for**. Vom începe prin recapitularea noţiunilor fundamentale, începând cu formele cele mai uzuale ale instrucţiunii **for**.

Forma generală a buclei **for** pentru repetarea unei singure instrucţiuni este:

for(*iniţializare; condiţie; iteraţie) instrucţiune;*

Pentru repetarea unui bloc, forma generală este:

for(*iniţializare; condiţie; iteraţie*)
{
  secvenţă de instrucţiuni
}

Iniţializarea este de regulă o instrucţiune de atribuire care fixează valoarea iniţială a *variabilei de control al buclei*, care este de cele mai multe ori un contor. *Condiţia* este o expresie de tipul **bool** care stabileşte dacă bucla mai continuă ciclarea. Expresia *iteraţie* stabileşte cantitatea cu care variabila de control al buclei se modifică la fiecare repetare a buclei. Remarcaţi că aceste trei secţiuni majore ale buclei **for** trebuie să fie separate prin punct şi virgulă. Bucla **for** continuă să se execute atât timp cât condiţia rămâne adevărată. Atunci când condiţia devine falsă, bucla se termină, iar execuţia programului continuă cu instrucţiunea care urmează imediat după **for**.

Programul de mai jos utilizează o buclă **for** pentru a afişa rădăcinile pătrate ale numerelor naturale de la 1 la 99. Programul afişează de asemenea şi erorile de rotunjire pentru fiecare rădăcină pătrată.

```
// Afisarea radicalilor numerelor de la 1 la 99 si a erorilor de rotunjire

using System;

class SqrRoot {
  public static void Main() {
    double num, sroot, rerr;

    for(num = 1.0; num < 100.0; num++) {
      sroot = Math.Sqrt(num);
      Console.WriteLine("Radical din " + num + " este " + sroot);

      // calculul erorii de rotunjire
      rerr = num - (sroot * sroot);
      Console.WriteLine("Eroarea de rotunjire este " + rerr);
      Console.WriteLine();
    }
  }
}
```

Observaţi că eroarea de rotunjire se calculează ridicând la pătrat rădăcina pătrată a fiecărui număr. Rezultatul se scade din numărul iniţial, determinând astfel eroarea de rotunjire. Desigur că, în unele cazuri, ridicarea la pătrat determină alte erori de rotunjire, ceea ce înseamnă că uneori însăşi eroarea de rotunjire este rotunjită !

Acest exemplu demonstrează faptul că operaţiile în virgulă mobilă nu sunt întotdeauna atât de precise pe cât ne aşteptăm.

Bucla **for** poate avansa cu pas pozitiv sau negativ, deoarece variabila de control al buclei se poate modifica cu orice cantitate. De exemplu, programul următor afişează numerele cuprinse între 100 şi –100, cu un pas negativ de lungime 5:

```
// O bucla for cu pasul negativ.

using System;

class DecrFor {
  public static void Main() {
    int x;

    for(x = 100; x > -100; x -=5)
      Console.WriteLine(x);
  }
}
```

Variabila de control al buclei este decrementată cu 5 de fiecare dată când se execută bucla.

Un aspect important legat de bucla **for** este că expresia condiţională este întotdeauna testată la începutul buclei. Aceasta înseamnă că secvenţa de cod din interiorul buclei nu se va executa deloc, atunci când condiţia este falsă de la început. Iată şi un exemplu:

```
for(count = 10; count < 5; count++)
  x += count;  // nu se executa
```

Această buclă nu se execută niciodată deoarece variabila sa de control, **count**, are o valoare mai mare de 5 la prima execuţie. Aceasta determină ca expresia condiţională **count < 5** să fie de la început falsă; în consecinţă, bucla nu va efectua nici o iteraţie.

Variante ale buclei for

Bucla **for** este una din cele mai flexibile instrucţiuni din limbajul C#, deoarece admite un număr mare de variante. Într-un prim exemplu, se pot utiliza mai multe variabile de control. Să considerăm următorul fragment de program:

```
// Utilizarea virgulelor intr-o instructiune for.

using System;

class Comma {
  public static void Main() {
    int i, j;

    for(i=0, j=10; i < j; i++, j--)
      Console.WriteLine("i si j: " + i + " " + j);
  }
}
```

Remarcaţi cele două variabile de control.

Textul afişat de program este prezentat mai jos:

```
i si j: 0 10
i si j: 1 9
i si j: 2 8
i si j: 3 7
i si j: 4 6
```


Aici, virgulele separă cele două instrucţiuni de iniţializare şi cele două expresii de iteraţie. La începutul buclei, se iniţializează ambele variabile **i** şi **j**. La fiecare repetare a buclei, **i** este incrementat, iar **j** decrementat. Variabilele de control multiple sunt de obicei comod de utilizat şi pot simplifica anumiţi algoritmi. Puteţi avea oricâte instrucţiuni de iniţializare şi de iteraţie, dar în practică utilizarea a mai mult de două va transforma bucla **for** într-o construcţie greoaie.

Condiţia care controlează bucla poate fi orice expresie corectă care produce un rezultat de tipul **bool**. Aceasta nu trebuie neapărat să conţină variabila de control al buclei. În exemplul următor, bucla continuă până când utilizatorul tastează litera S.

```
// Cicleaza pana la tastarea literei S.

using System;

class ForTest {
  public static void Main() {
    int i;

    Console.WriteLine("Apasati S pentru oprire.");

    for(i = 0; (char) Console.Read() != 'S'; i++)
      Console.WriteLine("Iteratia #" + i);
  }
}
```

Poziţii lipsă

Câteva variante interesante ale instrucţiunii **for** se obţin lăsând anumite părţi din definiţia buclei vide. În C#, este posibil ca oricare din părţile de iniţializare, condiţie şi iteraţie să fie vidă. De exemplu, să considerăm următorul program:

```
// Partile buclei for pot fi vide.

using System;

class Empty {
  public static void Main() {
    int i;

    for(i = 0; i < 10;) {
      Console.WriteLine("Pasul #" + i);
      i++; // incrementam variabila de control a buclei
    }
  }
}
```

Expresia de iteraţie lipseşte.

Expresia de iteraţie a buclei **for** este aici vidă. Variabila de control este acum incrementată în corpul buclei. Aceasta înseamnă că, la fiecare repetare a buclei, se verifică dacă **i** a ajuns la valoarea 10, fără a se incrementa însă variabila. Desigur că, deoarece variabila **i** este incrementată în corpul buclei, aceasta va rula corect, afişând următorul rezultat:

```
Pasul #0
Pasul #1
Pasul #2
Pasul #3
Pasul #4
Pasul #5
Pasul #6
Pasul #7
Pasul #8
Pasul #9
```

În exemplul următor, partea de inițializare este de asemenea mutată în afara buclei **for**.

```
// Mutam mai mult in afara buclei for.

using System;

class Empty2 {
  public static void Main() {
    int i;

    i = 0; // deplasam initializarea in afara buclei   <-- se mută expresia de inițializare afară din buclă
    for(; i < 10; ) {
      Console.WriteLine("Pasul #" + i);
      i++; // incrementam variabila de control a buclei
    }
  }
}
```

În această versiune, inițializarea lui **i** s-a făcut înainte de începerea buclei, în loc ca aceasta să facă parte din **for**. De regulă, veți inițializa variabila de control înăuntrul buclei **for**. Inițializarea se plasează în afara buclei numai atunci când valoarea inițială este rezultatul unui proces complicat, care nu se potrivește pentru a fi inclus în instrucțiunea **for**.

Bucle infinite

Puteți crea o buclă infinită (o buclă care nu se termină niciodată) utilizând instrucțiunea **for** cu expresia condițională vidă. De exemplu, secvența de mai jos prezintă maniera în care majoritatea programatorilor C# creează o buclă infinită:

```
for(;;) // bucla intentionat infinita
{
  //...
}
```

Această buclă ciclează la infinit. Chiar dacă există anumite programe, cum sunt interpretoarele de comenzi din sistemele de operare, care necesită o buclă infinită, majoritatea „buclelor infinite“ sunt în realitate doar bucle cu condiții speciale de terminare. Spre finalul acestui modul, veți vedea cum se poate opri o astfel de buclă. (Indicație: Se utilizează instrucțiunea **break**.)

Bucle fără corp

În C#, corpul asociat buclei **for** (și oricărei alte bucle) poate fi vid. Aceasta deoarece *instrucțiunea vidă* reprezintă o construcție corectă din punct de vedere sintactic. Buclele fără corp se dovedesc deseori utile. De exemplu, programul următor utilizează o astfel de buclă pentru însumarea numerelor de la 1 la 5:

```
// Corpul unei bucle poate fi vid.

using System;

class Empty3 {
  public static void Main() {
    int i;
    int sum = 0;

    // se insumeaza numerele de la 1 la 5
    for(i = 1; i <= 5; sum += i++);   <-- Nu există corp în această buclă!

    Console.WriteLine("Suma este " + sum);
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Suma este 15
```

Observați că procesul de însumare este efectuat complet în cadrul instrucțiunii **for** și corpul nu mai este necesar. Remarcați mai ales forma expresiei de iterație:

```
sum += i++
```

Nu trebuie să vă speriați de astfel de instrucțiuni. Ele apar frecvent în programele C# scrise de profesioniști, fiind ușor de înțeles dacă le separați în părțile componente. În cuvinte, instrucțiunea anterioară se traduce prin „adaugă la **sum** valoarea lui **i**, după care incrementează **i**“. Instrucțiunea este deci echivalentă cu secvența:

```
sum = sum + i;
i++;
```

Declararea variabilelor de control în interiorul buclei for

Adeseori, variabila care controlează o buclă **for** este necesară numai în cadrul buclei și nu mai este utilizată în altă parte. În această situație, este posibil să declarați variabila în partea de inițializare a lui **for**. De exemplu, programul următor calculează atât suma, cât și produsul numerelor de la 1 la 5. Variabila de control **i** este declarată înăuntrul buclei **for**.

```
// Declararea variabilei de control inauntrul lui for.

using System;
```

```
class ForVar {
  public static void Main() {
    int sum = 0;
    int fact = 1;

    // calculeaza suma si produsul numerelor de la 1 la 5
    for(int i = 1; i <= 5; i++) {
      sum += i;
      fact *= i;
    }

    // i nu este cunoscut aici.

    Console.WriteLine("Suma este " + sum);
    Console.WriteLine("Produsul este " + fact);
  }
}
```

Variabila **i** este declarată în interiorul instrucțiunii **for**.

La declararea unei variabile înăuntrul buclei **for**, este bine să rețineți un aspect: domeniul de vizibilitate al acelei variabile se termină acolo unde se termină instrucțiunea **for**. (Adică, domeniul variabilei este limitat la bucla **for**.) În afara buclei **for**, variabila încetează să mai existe. Așadar, în exemplul precedent, **i** nu este accesibil în afara buclei **for**. Dacă aveți nevoie să utilizați variabila de control a buclei și în altă parte din program, nu o puteți declara înăuntrul buclei **for**.

Înainte de a trece mai departe, puteți experimenta propriile variante de bucle **for**. Vă veți da seama că instrucțiunea **for** este absolut fascinantă.

Exerciții la minut

- Este posibil ca părțile buclei **for** să fie vide ?
- Arătați cum puteți crea o buclă infinită utilizând **for**.
- Care este domeniul unei variabile declarate într-o instrucțiune **for** ?

Bucla while

O altă buclă în C# este **while**. Forma generală a buclei **while** este:

while(*condiție*) *instrucțiune*;

unde *instrucțiune* poate fi o singură instrucțiune sau un bloc, iar *condiție* definește condiția care controlează bucla și poate fi orice expresie booleană corectă. Instrucțiunea se execută atât timp cât condiția este adevărată. Când condiția devine falsă, controlul trece la linia din program imediat următoare buclei.

- Da. Toate cele trei părți ale lui **for**: inițializarea, condiția și iterația pot fi vide.
- **for(;;)**
- Domeniul unei variabile declarate într-o instrucțiune **for** se limitează la bucla **for**. În afara acesteia, variabila nu este cunoscută.

Iată un exemplu simplu în care bucla **while** se utilizează pentru afișarea literelor alfabetului:

```
// Prezentarea buclei while.

using System;

class WhileDemo {
  public static void Main() {
    char ch;

    // afiseaza alfabetul utilizand o bucla while
    ch = 'a';
    while(ch <= 'z') {
      Console.Write(ch);
      ch++;
    }
  }
}
```

Aici, **ch** a fost inițializat cu litera a. La fiecare pas al buclei, **ch** este afișat și apoi incrementat. Operația continuă până când **ch** devine mai mare decât z.

Ca și bucla **for**, **while** verifică expresia condițională la început, deci este posibil ca secvența din corpul buclei să nu se execute deloc. Aceasta elimină necesitatea de a efectua un test preliminar înainte de începutul buclei. Programul următor demonstrează această caracteristică a buclei **while**. Programul calculează puterile întregi cuprinse între 0 și 9 ale lui 2.

```
// Calculul puterilor intregi ale lui 2.

using System;

class Power {
  public static void Main() {
    int e;
    int result;

    for(int i = 0; i < 10; i++) {
      result = 1;
      e = i;
      while(e > 0) {
        result *= 2;
        e--;
      }

      Console.WriteLine("2 la puterea " + i + " este " + result);
    }
  }
}
```

Rezultatul afișat de program este prezentat în continuare:

```
2 la puterea 0 este 1
2 la puterea 1 este 2
2 la puterea 2 este 4
2 la puterea 3 este 8
```

```
2 la puterea 4 este 16
2 la puterea 5 este 32
2 la puterea 6 este 64
2 la puterea 7 este 128
2 la puterea 8 este 256
2 la puterea 9 este 512
```

Observați că bucla **while** se execută numai dacă **e** este strict pozitiv. Prin urmare, când **e** este 0, ca de exemplu la prima iterație a buclei **for**, bucla **while** nu se execută.

Sfatul expertului

Întrebare: Cunoscând flexibilitatea inerentă a tuturor buclelor din C#, ce criterii trebuie folosite la alegerea uneia dintre instrucțiunile repetitive ? Cum se poate deci selecta bucla cea mai adecvată pentru o anumită prelucrare ?

Răspuns: Utilizați o buclă **for** atunci când efectuați un număr cunoscut de iterații. Utilizați **do-while** dacă aveți nevoie ca bucla să execute cel puțin o iterație. Instrucțiunea **while** se recomandă atunci când prelucrarea presupune un număr necunoscut de pași.

Bucla do-while

Ultima dintre instrucțiunile repetitive din C# este **do-while**. Spre deosebire de **for** și **while**, la care condiția se testează la începutul buclei, **do-while** își testează condiția la sfârșit. Aceasta înseamnă că orice buclă **do-while** se execută cel puțin o dată. Forma generală a buclei **do-while** este:

```
do {
  instrucțiuni;
} while(condiție);
```

Chiar dacă acoladele nu sunt necesare atunci când avem o singură instrucțiune, ele se includ în multe cazuri pentru a face construcția **do-while** mai ușor de citit, evitând astfel confuzia cu **while**. Bucla **do-while** se execută atât timp cât expresia condițională rămâne adevărată.

Programul următor ciclează până când utilizatorul tastează litera **q**:

```
// Prezentarea buclei do-while.

using System;

class DWDemo {
  public static void Main() {
    char ch;

    do {
      Console.Write("Apasati o tasta si apoi ENTER: ");
      ch = (char) Console.Read();   // citeste // un caracter
    } while(ch != 'q');
  }
}
```


Utilizând o buclă **do-while**, putem îmbunătăți programul pentru jocul „ghicește litera" de la începutul acestui modul. De data aceasta, programul va cicla până când utilizatorul ghicește litera.

```
// "Ghiceste litera", versiunea a patra

using System;

class Guess4 {
  public static void Main() {
    char ch, answer = 'K';

    do {
      Console.WriteLine("Ma gandesc la o litera de la A la Z.");
      Console.Write("O poti ghici: ");

      // citeste o litera, ignorand CR/LF
      do {
        ch = (char) Console.Read();  // citeste un caracter
      } while(ch == '\n' | ch == '\r');

      if(ch == answer) Console.WriteLine("** Corect **");
      else {
        Console.Write("...Imi pare rau, litera e ");
        if(ch < answer) Console.WriteLine("prea mica.");
        else Console.WriteLine("prea mare.");
        Console.WriteLine("Incearca inca o data!\n");
      }
    } while(answer != ch);
  }
}
```

Iată o rulare de probă a programului:

```
Ma gandesc la o litera de la A la Z.
O poti ghici: A
...Imi pare rau, litera e prea mica.
Incearca inca o data!

Ma gandesc la o litera de la A la Z.
O poti ghici: Z
...Imi pare rau, litera e prea mare.
Incearca inca o data!

Ma gandesc la o litera de la A la Z.
O poti ghici: K
** Corect **
```

Observați un alt aspect interesant în acest program. Bucla **do-while** de mai jos citește următorul caracter, ignorând însă caracterele „revenire la cap de rând" (CR) și „linie nouă" (LF) din fluxul de intrare:

```
// citeste o litera, ignorand CR/LF
do {
  ch = (char) Console.Read(); // citeste un caracter
} while(ch == '\n' | ch == '\r');
```

Iată de ce este această buclă necesară. După cum s-a precizat mai devreme, intrarea de la consolă este direcționată într-un buffer de linie – adică trebuie să apăsați ENTER pentru a trimite caracterele tastate către program. Apăsarea tastei ENTER determină însă generarea caracterelor „revenire la cap de rând“ și „linie nouă“. Aceste caractere rămân în buffer-ul de intrare. Bucla elimină aceste caractere, citind din buffer-ul de intrare până când acesta rămâne gol.

Exerciții la minut

- Care este principala diferență dintre buclele **while** și **do-while** ?
- Condiția care controlează instrucțiunea **while** poate fi de orice tip. Este adevărat sau fals ?

Proiectul 3-2 Îmbunătățirea sistemului de help pentru C#

Acest proiect va dezvolta sistemul de help pentru C# creat în cadrul proiectului 3-1. Această versiune adaugă sintaxa buclelor **for**, **while** și **do-while**. De asemenea, programul verifică selecția utilizatorului din meniul afișat inițial, rulând în buclă până când acesta introduce un răspuns corect.

Pas cu pas

1. Copiați conținutul fișierului **Help.cs** într-un fișier nou, numit **Help2.cs**.
2. Modificați partea în care programul afișează opțiunile, incluzând bucla prezentată mai jos:

```
do {
  Console.WriteLine("Help la:");
  Console.WriteLine("  1. if");
  Console.WriteLine("  2. switch");
  Console.WriteLine("  3. for");
  Console.WriteLine("  4. while");
  Console.WriteLine("  5. do-while\n");
  Console.WriteLine("Alegerea dumneavoastra: ");
  do {
    choice = (char) Console.Read();
  } while(choice == '\n' | choice == '\r');
} while(choice < '1' | choice > '5');
```

Remarcați utilizarea unei bucle **do-while** imbricate pentru îndepărtarea caracterelor „revenire la cap de rând“ și „linie nouă“ nedorite din stream-ul de intrare. După această modificare, programul va cicla afișând meniul până când utilizatorul va introduce un răspuns între 1 și 5.

3. Extindeți instrucțiunea **switch** pentru a cuprinde și buclele **for**, **while** și **do-while**, după cum se observă mai jos:

```
switch(choice) {
  case '1':
    Console.WriteLine("Instructiunea if:\n");
    Console.WriteLine("if(conditie) instructiune;");
    Console.WriteLine("else instructiune;");
    break;
  case '2':
    Console.WriteLine("Instructiunea switch:\n");
    Console.WriteLine("switch(expresie) {");
    Console.WriteLine("  case constanta:");
    Console.WriteLine("  secventa de instructiuni");
    Console.WriteLine("  break;");
    Console.WriteLine("  // ...");
    Console.WriteLine("}");
    break;
  case '3':
    Console.WriteLine("Instructiunea for:\n");
    Console.WriteLine("for(init; conditie; iteratie)");
    Console.WriteLine(" instructiune;");
    break;
  case '4':
    Console.WriteLine("Instructiunea while:\n");
    Console.WriteLine("while(conditie) instructiune;");
    break;
  case '5':
    Console.WriteLine("Instructiunea do-while:\n");
    Console.WriteLine("do {");
    Console.WriteLine(" instructiune;");
    Console.WriteLine("} while(conditie);");
    break;
}
```

Observați că, în această versiune, instrucțiunea **switch** nu are o clauză **default**. Deoarece bucla care afișează meniul garantează introducerea unui răspuns corect, nu mai este necesar să includem o clauză **default** care să trateze opțiunile incorecte.

4. Iată listingul complet al programului **Help2.cs**:

```
/*
  Proiectul 3-2

  Un sistem de help imbunatatit care utilizeaza o instructiune do-while
  pentru a prelucra selectia din meniu.
*/

using System;

class Help2 {
  public static void Main() {
    char choice;

    do {
      Console.WriteLine("Help la:");
      Console.WriteLine("  1. if");
```

- Bucla **while** își testează condiția la început. Bucla **do-while** o testează la sfârșit. În consecință, **do-while** se execută cel puțin o dată.
- Fals. Condiția trebuie să fie de tipul **bool**.

```
      Console.WriteLine("   2. switch");
      Console.WriteLine("   3. for");
      Console.WriteLine("   4. while");
      Console.WriteLine("   5. do-while\n");
      Console.WriteLine("Alegerea dumneavoastra: ");
      do {
        choice = (char) Console.Read();
      } while(choice == '\n' | choice == '\r');
    } while(choice < '1' | choice > '5');

    Console.WriteLine("\n");

    switch(choice) {
      case '1':
        Console.WriteLine("Instructiunea if:\n");
        Console.WriteLine("if(conditie) instructiune;");
        Console.WriteLine("else instructiune;");
        break;
      case '2':
        Console.WriteLine("Instructiunea switch:\n");
        Console.WriteLine("switch(expresie) {");
        Console.WriteLine("  case constanta:");
        Console.WriteLine("secventa de instructiuni");
        Console.WriteLine("  break;");
        Console.WriteLine("  // ...");
        Console.WriteLine("}");
        break;
      case '3':
        Console.WriteLine("Instructiunea for:\n");
        Console.WriteLine("for(init; conditie; iteratie)");
        Console.WriteLine(" instructiune;");
        break;
      case '4':
        Console.WriteLine("Instructiunea while:\n");
        Console.WriteLine("while(conditie) instructiune;");
        break;
      case '5':
        Console.WriteLine("Instructiunea do-while:\n");
        Console.WriteLine("do {");
        Console.WriteLine(" instructiune;");
        Console.WriteLine("} while(conditie);");
        break;
    }
  }
}
```

Utilizarea instrucțiunii **break** pentru părăsirea unei bucle

Este posibil să forțăm părăsirea imediată a unei instrucțiuni repetitive, ignorând codul rămas în corpul buclei și testul condițional al acesteia prin utilizarea instrucțiunii **break**. Atunci când instrucțiunea **break** este întâlnită în interiorul unei

bucle, aceasta se termină, iar execuția programului continuă cu instrucțiunea care urmează imediat după buclă. Un exemplu simplu este prezentat mai jos:

```
// Utilizarea unui break pentru parasirea unei bucle.

using System;

class BreakDemo {
  public static void Main() {
    int num;

    // cicleaza cat timp patratul lui i este mai mic decat num
    for(int i = 0; i < num; i++) {
      if(i*i >= num) break;  // iese din bucla daca i*i >= 100
      Console.Write(i + " ");
    }
    Console.WriteLine("Bucla s-a terminat.");
  }
}
```

Folosiți **break** pentru a termina bucla.

Programul afișează următorul text:

```
0 1 2 3 4 5 6 7 8 9 Bucla s-a terminat.
```

După cum puteți observa, chiar dacă inițial s-a intenționat ca bucla **for** să ruleze între 0 și **num** (care este 100 în acest caz), instrucțiunea **break** face ca aceasta să se termine mai devreme, de îndată ce pătratul lui **i** devine cel puțin egal cu **num**.

Instrucțiunea **break** poate fi utilizată în conjuncție cu oricare dintre buclele din C#, inclusiv cu cele intenționat infinite. De exemplu, următorul program citește date introduse de la tastatură, până când utilizatorul apasă tasta **q**.

```
// Citeste de la tastatura, pana cand se introduce q.

using System;

class Break2 {
  public static void Main() {
    char ch;

    for( ; ; ) {
      ch = (char) Console.Read();  // citeste un caracter
      if(ch == 'q') break;
    }
    Console.WriteLine("Ati apasat q!");
  }
}
```

Această buclă „infinită" este terminată de către **break**.

Atunci când este utilizată în interiorul unui set de bucle imbricate, instrucțiunea **break** determină ieșirea din bucla cea mai interioară. De exemplu:

```
// Utilizarea lui break pentru bucle imbricate.

using System;

class Break3 {
  public static void Main() {
```

```
    for(int i=0; i<3; i++) {
      Console.WriteLine("Contorul buclei exterioare: " + i);
      Console.WriteLine("   Contorul buclei interioare: ");

      int t = 0;
      while(t < 100) {
        if(t == 10) break; // iesire din bucla daca t este 10
        Console.Write(t + " ");
        t++;
      }
      Console.WriteLine();
    }
    Console.WriteLine("Bucle terminate.");
  }
}
```

Sfatul expertului

Întrebare: Ştiu că în Java, instrucţiunile `break` şi `continue` pot fi utilizate împreună cu o etichetă. Există şi în C# această facilitate ?

Răspuns: Nu. Cei care au proiectat limbajul C# nu au înzestrat instrucţiunile **break** şi **continue** cu această posibilitate. Acestea funcţionează în C# la fel ca în C şi C++. Motivul principal pentru care C# nu a urmat exemplul limbajului Java este acela că Java nu dispune de instrucţiunea **goto**, care este însă prevăzută în C#. Limbajul Java are deci nevoie de această facilitate suplimentară pentru **break** şi **continue** pentru a suplini lipsa lui **goto**.

Programul generează următorul text:

```
Contorul buclei exterioare: 0
   Contorul buclei interioare: 0 1 2 3 4 5 6 7 8 9
Contorul buclei exterioare: 1
   Contorul buclei interioare: 0 1 2 3 4 5 6 7 8 9
Contorul buclei exterioare: 2
   Contorul buclei interioare: 0 1 2 3 4 5 6 7 8 9
Bucle terminate.
```

După cum se observă, instrucţiunea **break** din bucla interioară determină ieşirea numai din acea buclă. Bucla exterioară nu este afectată.

Iată alte două aspecte de reţinut despre **break**. Mai întâi, într-o buclă pot apărea mai multe instrucţiuni **break**. Oricum, este cazul să fiţi precaut. Prea multe instrucţiuni **break** au tendinţa de a submina structurarea codului. În al doilea rând, un **break** care termină o instrucţiune **switch** afectează numai acea instrucţiune, nu şi eventualele bucle care o conţin.

Utilizarea instrucţiunii continue

Există posibilitatea de a forţa efectuarea mai devreme a unei iteraţii într-o buclă, evitând structura normală de control a buclei. Aceasta se realizează utilizând instrucţiunea **continue**. Instrucţiunea **continue** forţează executarea următoarei iteraţii a buclei, sărind peste codul rămas până la sfârşitul iteraţiei curente. **Continue** reprezintă în esenţă instrucţiunea complementară lui **break**. Ca exemplu, programul următor utilizează **continue** pentru afişarea numerelor pare cuprinse între 0 şi 100:

```
// Utilizarea lui continue.

using System;

class ContDemo {
  public static void Main() {
    int i;

    // afiseaza numerele pare dintre 0 si 100
    for(i = 0; i <= 100; i++) {
      if((i % 2) != 0) continue;  // trecere la iteratia urmatoare
      Console.WriteLine(i);
    }
  }
}
```

Continuă dacă **i** este impar.

Se afişează numai numerele pare, deoarece cele impare determină ca bucla să treacă mai devreme la iteraţia următoare, sărind peste apelul lui **WriteLine()**.

În buclele **while** şi **do-while**, instrucţiunea **continue** determină trecerea controlului direct la expresia condiţională, continuând apoi procesul repetitiv. În cazul lui **for**, se evaluează expresia de iteraţie, se execută expresia condiţională, după care bucla continuă.

Beneficiile reale aduse de instrucţiunea **continue** sunt rare. Motivul principal este că limbajul C# pune la dispoziţie un set bogat de instrucţiuni repetitive, care satisfac cerinţele majorităţii aplicaţiilor. Instrucţiunea **continue** reprezintă o modalitate structurată de rezolvare a cazurilor cu totul speciale care necesită forţarea unei noi iteraţii.

Instrucţiunea goto

Goto este instrucţiunea de salt necondiţionat în C#. La întâlnirea acesteia, fluxul execuţiei programului se transferă la locaţia specificată de **goto**. Această instrucţiune a căzut în dizgraţia programatorilor în urmă cu mulţi ani, deoarece încurajează crearea codului „spaghetti". Cu toate acestea, instrucţiunea **goto** încă se mai utilizează în mod ocazional – câteodată fiind chiar eficientă. Lucrarea de faţă nu va da o sentinţă referitoare la justificarea acestei instrucţiuni ca structură de control într-un program. Trebuie însă să stabilim că nu există situaţii în programare care să oblige la utilizarea instrucţiunii **goto** – aceasta nefiind un element necesar pentru completitudinea unui limbaj de programare. Mai degrabă, această instrucţiune este o convenţie care, dacă

este utilizată chibzuit, poate aduce avantaje în anumite situații. În concluzie, nu vom utiliza **goto** în această carte în afara acestei secțiuni. Grija principală pe care majoritatea programatorilor trebuie să o manifeste față de **goto** este legată de tendința acestei instrucțiuni de a dezorganiza programele și de a le face aproape imposibil de citit. Există însă câteva situații în care utilizarea lui **goto** are menirea de a simplifica execuția unui program și nu de a o dezorganiza.

Instrucțiunea **goto** necesită utilizarea unei etichete. O *etichetă* este un identificator valid C# urmat de două puncte (:). Mai mult, eticheta trebuie să fie în aceeași metodă cu instrucțiunea **goto** care o utilizează. De exemplu, o buclă cu 100 de iterații se poate scrie utilizând **goto** și o etichetă după cum urmează:

```
x = 1;
loop1: ◄──────────────────────┐
  x++;                                   │
  if(x < 100) goto loop1; ──► Execuția sare la loop1.
```

Una dintre utilizările benefice pentru **goto** apare la părăsirea unei rutine cu nivel mare de imbricare. Iată un exemplu simplu.

```
/*
Prezentarea lui goto.
  */

using System;

class Use_goto {
  public static void Main() {
    int i=0, j=0, k=0;

    for(i=0; i < 10; i++) {
      for(j=0; j < 10; j++) {
        for(k=0; k < 10; k++) {
          Console.WriteLine("i, j, k: ", + i + " " + j + " " + k);
          if(k == 3) goto stop;
        }
      }
    }

stop:
    Console.WriteLine("Terminat! i, j, k: " + i + ", " + j + " " + k);
  }
}
```

Rezultatul afișat de acest program este prezentat mai jos:

```
i, j, k: 0 0 0
i, j, k: 0 0 1
i, j, k: 0 0 2
i, j, k: 0 0 3
Terminat! i, j, k: 0, 0 3
```

Renunțarea la **goto** ar forța utilizarea a câte trei instrucțiuni **if** și **break**. În acest caz, **goto** a simplificat codul. Acesta este doar un exemplu conceput în scop demonstrativ, dar vă puteți gândi și la alte situații în care **goto** să fie de ajutor.

Exerciții la minut

- Ce se întâmplă într-o buclă atunci când se execută o instrucțiune **break** ?
- Ce rol are **continue** ?
- Care este instrucțiunea de salt necondiționat din C# ?

Proiectul 3-3: Definitivarea sistemului de help pentru C#

Acest proiect efectuează ultimele retușuri la sistemul de help pentru C# creat în cadrul proiectelor anterioare. Noua versiune adaugă sintaxa pentru **break**, **continue** și **goto**. Tot această versiune va permite utilizatorului să vizualizeze sintaxa mai multor instrucțiuni. Aceasta se realizează adăugând o buclă exterioară care se execută până când utilizatorul introduce litera **q** pe post de selecție din meniu.

Pas cu pas

1. Copiați conținutul fișierului **Help2.cs** într-un fișier nou, numit **Help3.cs**.
2. Includeți tot codul programului într-o buclă **for** infinită. Asigurați ieșirea din această buclă cu o instrucțiune **break**, atunci când se introduce litera **q**. Deoarece această buclă include întreg codul programului, părăsirea ei conduce la terminarea programului.
3. Modificați bucla care afișează meniul după cum urmează:

```
do {
  Console.WriteLine("Help la:");
  Console.WriteLine("  1. if");
  Console.WriteLine("  2. switch");
  Console.WriteLine("  3. for");
  Console.WriteLine("  4. while");
  Console.WriteLine("  5. do-while\n");
  Console.WriteLine("  6. break\n");
  Console.WriteLine("  7. continue\n");
  Console.WriteLine("  8. goto\n");
  Console.WriteLine("Alegerea dumneavoastra (q pentru terminare): ");
  do {
    choice = (char) Console.Read();
  } while(choice == '\n' | choice == '\r');
} while(choice < '1' | choice > '8' & choice != 'q');
```

   Remarcați că această buclă cuprinde acum și instrucțiunile **break**, **continue** și **goto**. De asemenea, acceptă litera **q** ca opțiune validă.
4. Extindeți instrucțiunea **switch** pentru a cuprinde și instrucțiunile **break**, **continue** și **goto**, după cum urmează:

```
case '6':
  Console.WriteLine("Instructiunea break:\n");
```

- Într-o buclă, **break** cauzează terminarea imediată a buclei. Execuția continuă cu prima instrucțiune de după buclă.
- Instrucțiunea **continue** determină o buclă să treacă imediat la iterația următoare, sărind peste codul rămas.
- Instrucțiunea de salt necondiționat din C# este **goto**.

```
      Console.WriteLine("break;");
      break;
    case '7':
      Console.WriteLine("Instructiunea continue:\n");
      Console.WriteLine("continue;");
      break;
    case '8':
      Console.WriteLine("Instructiunea goto:\n");
      Console.WriteLine("goto eticheta;");
      break;
```

5. Iată listingul complet al programului `Help3.cs`:

```
/*
   Proiectul 3-3

   Versiunea finala a sistemului de help pentru instructiunile C#,
   care afiseaza informatii despre mai multe instructiuni.
*/

using System;

class Help3 {
  public static void Main() {
    char choice;

    for( ; ; ) {
      do {
        Console.WriteLine("Help la:");
        Console.WriteLine("  1. if");
        Console.WriteLine("  2. switch");
        Console.WriteLine("  3. for");
        Console.WriteLine("  4. while");
        Console.WriteLine("  5. do-while\n");
        Console.WriteLine("  6. break\n");
        Console.WriteLine("  7. continue\n");
        Console.WriteLine("  8. goto\n");
        Console.WriteLine("Alegerea dumneavoastra (q pentru terminare): ");
        do {
          choice = (char) Console.Read();
        } while(choice == '\n' | choice == '\r');
      } while(choice < '1' | choice > '8' & choice != 'q');

      if(choice == 'q') break;

      Console.WriteLine("\n");

      switch(choice) {
        case '1':
          Console.WriteLine("Instructiunea if:\n");
          Console.WriteLine("if(conditie) instructiune;");
          Console.WriteLine("else instructiune;");
          break;
        case '2':
          Console.WriteLine("Instructiunea switch:\n");
          Console.WriteLine("switch(expresie) {");
          Console.WriteLine("  case constanta:");
          Console.WriteLine("secventa de instructiuni");
          Console.WriteLine("  break;");
          Console.WriteLine("  // ...");
          Console.WriteLine("}");
          break;
        case '3':
          Console.WriteLine("Instructiunea for:\n");
          Console.WriteLine("for(init; conditie; iteratie)");
          Console.WriteLine(" instructiune;");
          break;
        case '4':
          Console.WriteLine("Instructiunea while:\n");
          Console.WriteLine("while(conditie) instructiune;");
          break;
        case '5':
          Console.WriteLine("Instructiunea do-while:\n");
          Console.WriteLine("do {");
          Console.WriteLine(" instructiune;");
          Console.WriteLine("} while(conditie);");
          break;
        case '6':
          Console.WriteLine("Instructiunea break:\n");
          Console.WriteLine("break;");
          break;
        case '7':
          Console.WriteLine("Instructiunea continue:\n");
          Console.WriteLine("continue;");
          break;
        case '8':
          Console.WriteLine("Instructiunea goto:\n");
          Console.WriteLine("goto eticheta;");
          break;
      }
      Console.WriteLine();
    }
  }
}
```


Iată un posibil dialog al programului cu utilizatorul:

```
Help la:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue
  8. goto
Alegerea dumneavoastra (q pentru terminare): 1

Instructiunea if:

if(conditie) instructiune;
else instructiune;

Help la:
  1. if
  2. switch
```

```
    3. for
    4. while
    5. do-while
    6. break
    7. continue
    8. goto

 Alegerea dumneavoastra (q pentru terminare): 6

 Instructiunea break:

 break;

 Help la:
    1. if
    2. switch
    3. for
    4. while
    5. do-while
    6. break
    7. continue
    8. goto

 Alegerea dumneavoastra (q pentru terminare): q
```

Bucle imbricate

După cum ați observat din exemplele anterioare, o buclă poate fi imbricată într-o alta. Buclele imbricate se utilizează la rezolvarea unei game largi de probleme de programare, fiind o parte esențială a științei programării. Din acest motiv, înainte de a epuiza subiectul instrucțiunilor repetitive din C#, vom mai examina încă un exemplu în care apar bucle imbricate. Programul următor utilizează două bucle **for** imbricate pentru determinarea divizorilor proprii ai numerelor de la 2 la 100:

```
/*
   Utilizarea buclelor imbricate la determinarea divizorilor
   numerelor de la 2 la 100.
 */

using System;

class FindFac {
  public static void Main() {

    for(int i=2; i <= 100; i++) {
      Console.Write("Divizorii lui " + i + ": ");
      for(int j=2; j < i; j++)
        if((i%j) == 0) Console.Write(j + " ");
      Console.WriteLine();
    }
  }
}
```

În continuare, prezentăm o parte din rezultatul afişat de acest program:


```
Divizorii lui 2:
Divizorii lui 3:
Divizorii lui 4: 2
Divizorii lui 5:
Divizorii lui 6: 2 3
Divizorii lui 7:
Divizorii lui 8: 2 4
Divizorii lui 9: 3
Divizorii lui 10: 2 5
Divizorii lui 11:
Divizorii lui 12: 2 3 4 6
Divizorii lui 13:
Divizorii lui 14: 2 7
Divizorii lui 15: 3 5
Divizorii lui 16: 2 4 8
Divizorii lui 17:
Divizorii lui 18: 2 3 6 9
Divizorii lui 19:
Divizorii lui 20: 2 4 5 10
```

În acest program, bucla exterioară merge de la 2 până la 100. Bucla interioară testează la rând toate numerele de la 2 până la **i**, afişând acele numere la care **i** se împarte exact.

Verificarea cunoştinţelor

1. Scrieți un program care citește caractere de la tastatură până când se introduce un punct. Modificați programul astfel încât să numere spațiile introduse, afişând la sfârşit numărul lor.
2. La instrucțiunea **switch**, secvența de cod de la un **case** se poate continua cu următorul ?
3. Prezentați forma generală a scării **if-else-if**.
4. Se dă secvența:

```
if(x < 10)
  if(y > 100) {
    if(! done) x = z;
    else y = z;
  }
else Console.WriteLine("eroare"); // care if ?
```

   Precizați cu care **if** se asociază ultimul **else**.
5. Scrieți instrucțiunea **for** pentru o buclă care numără de la 1 000 la 0 cu pasul –2.
6. Este corectă secvența următoare ?

```
for(int i = 0; i < num; i++)
  sum += i;

count = i;
```

7. Explicați care este rolul instrucțiunii **break**.

8. În fragmentul următor, ce se afişează după execuţia instrucţiunii `break` ?

```
for(i = 0; i < 10; i++) {
  while(running) {
    if(x < y) break;
    //...
  }
  Console.WriteLine("dupa while");
}
Console.WriteLine("dupa for");
```

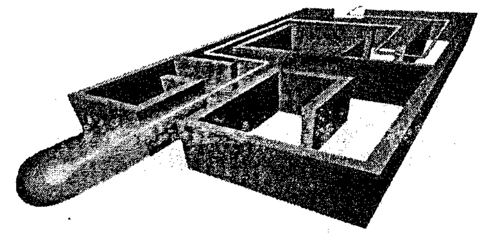
9. Ce afişează următorul fragment de cod ?

```
for(int i = 0; i < 10; i++) {
  Console.Write(i + " ");
  if((i%2) == 0) continue;
  Console.WriteLine();
}
```

10. Expresia de iteraţie dintr-o buclă `for` nu trebuie întotdeauna să modifice variabila de control a buclei cu o cantitate fixă. Variabila respectivă se poate modifica în mod arbitrar. Folosind acest concept, scrieţi un program care utilizează o buclă `for` pentru generarea şi afişarea progresiei geometrice 1, 2, 4, 8, 16, 32 etc.
11. Literele mici ale setului ASCII diferă de literele mari cu 32. Prin urmare, pentru a converti o literă mică în majusculă, se scade 32 din valoarea literei. Utilizaţi această informaţie pentru a scrie un program care citeşte caractere de la tastatură, schimbând literele mici în litere mari şi invers, afişând totodată rezultatul modificării. Celelalte caractere nu vor fi modificate. Programul va citi caractere până când utilizatorul tastează un punct. La final, programul va afişa numărul modificărilor efectuate.

Modulul 4

Prezentarea claselor, obiectelor şi metodelor



Scopurile acestui modul

- **Noţiuni fundamentale despre clase**
- **Crearea obiectelor**
- **Crearea metodelor**
- **Adăugarea parametrilor unei metode**
- **Întoarcerea unei valori dintr-o metodă**
- **Utilizarea constructorilor**
- **Operatorul `new` şi colectarea automată a obiectelor neutilizate**
- **Rolul destructorilor**
- **O privire generală asupra cuvântului cheie `this`**

Înainte de a avansa în studiul limbajului C#, trebuie să stăpâniţi noţiunea de clasă. Clasele sunt chintesenţa limbajului C#, deoarece definesc natura obiectelor. Clasele reprezintă fundamentul peste care este construit întreg limbajul C#. În concluzie, clasele oferă baza programării orientate spre obiecte în C#. În interiorul unei clase, sunt definite datele şi codul care prelucrează acele date. Codul este conţinut în metode. Deoarece clasele, obiectele şi metodele sunt aspecte esenţiale în C#, ele sunt prezentate în acest modul. Înţelegerea aspectelor de bază legate de aceste facilităţi vă va permite să dezvoltaţi programe mai complicate şi să vă însuşiţi mai bine alte elemente esenţiale ale limbajului C# prezentate în modulul următor.

Noţiuni fundamentale despre clase

Deoarece întreaga activitate a unui program C# se desfăşoară în cadrul unei clase, am utilizat clase încă de la începutul acestei cărţi. Desigur că s-au utilizat numai clase extrem de simple, fără a beneficia de majoritatea facilităţilor pe care clasele le oferă. După cum veţi vedea, clasele sunt cu mult mai puternice decât exemplele limitate prezentate până acum.

Să începem cu recapitularea unor aspecte fundamentale. O clasă reprezintă un şablon care defineşte forma unui obiect. Clasa specifică atât datele, cât şi codul care va prelucra acele date. Limbajul C# utilizează specificaţiile claselor pentru a construi

obiecte. Obiectele sunt *instanțe* ale claselor. Ca urmare, o clasă este de fapt un set de planuri care precizează cum se poate construi un obiect. Este foarte important să înțelegeți cât se poate de clar următoarea chestiune: o clasă este doar o noțiune abstractă. Până când nu se creează o instanță a clasei, aceasta nu dispune de nici o reprezentare fizică în memorie.

Un alt aspect: amintiți-vă că metodele și variabilele care alcătuiesc o clasă sunt denumite și *membri* ai clasei.

Formatul general al unei clase

Atunci când definiți o clasă, declarați datele pe care aceasta le conține și codul care prelucrează acele date. În timp ce clasele foarte simple pot conține numai cod sau numai date, majoritatea claselor din programele reale le conțin pe amândouă.

În linii mari, datele sunt conținute de variabilele instanță definite în cadrul clasei, iar codul se găsește în metode. Este important să precizăm însă de la început că limbajul C# poate defini mai multe categorii particulare de date și membri care conțin cod, printre care variabilele instanță, variabilele statice, constantele, metodele, constructorii, destructorii, indexările, evenimentele și proprietățile. Pentru moment, vom limita discuția noastră despre clase la elementele esențiale: variabilele instanță și metodele. Ulterior în acest modul, vom discuta despre constructori și destructori. Celelalte categorii de membri sunt prezentate în module ulterioare.

Clasele se creează utilizând cuvântul cheie **class**. Formatul general al unei definiții de clasă care conține numai variabile instanță și metode este prezentat mai jos:

```
class numeclasa {
   // declaratia variabilelor instanta
   acces tip var1;
   acces tip var2;
   // ...
   acces tip varN;

   // declaratia metodelor
   acces tip-rez metoda1(parametri) {
      // corpul metodei
   }
   acces tip-rez metoda2(parametri) {
      // corpul metodei
   }
   // ...
   acces tip-rez metodaN(parametri) {
      // corpul metodei
   }
}
```

Remarcați că toate variabilele și metodele sunt precedate de *acces*. *Acces* reprezintă aici un specificator de acces, cum ar fi **public**, care determină modul în care poate fi

referit fiecare membru în parte. După cum s-a menționat în modulul 1, membrii unei clase pot fi privați, sau pot fi mai accesibili. Specificatorul de acces determină tipul accesului permis asupra membrului. Specificatorii de acces sunt opționali, în lipsa acestora, membrii corespunzători fiind implicit membri privați ai clasei. Vom reveni la subiectul specificatorilor de acces într-un modul ulterior.

O clasă bine gândită trebuie să definească o singură entitate logică, deși nu există nici o regulă de sintaxă care să impună această corespondență. De exemplu, o clasă care memorează nume și numere de telefon nu va memora de regulă și informații despre piața bursieră, cantitățile medii de precipitații, ciclurile activității solare sau orice alte informații colaterale. Concluzia este că o clasă bine gândită trebuie să grupeze numai informații coerente din punct de vedere logic. Introducerea unor informații irelevante într-o clasă conduce rapid la pierderea structurării codului.

Până acum, toate clasele pe care le-am utilizat au avut o singură metodă: **Main()**. În curând, veți vedea cum se pot crea și alte metode. Observați însă că formatul general al unei clase nu specifică metoda **Main()**. Prezența metodei **Main()** este obligatorie numai în clasa care reprezintă punctul de pornire al programului dumneavoastră.

Definirea unei clase

Pentru a prezenta conceptul de clasă, vom dezvolta o clasă care încapsulează informații despre vehicule, ca de exemplu mașini, furgonete și camioane. Clasa este numită **Vehicle** și memorează trei informații despre un vehicul: numărul pasagerilor pe care acesta îl poate transporta, capacitatea rezervorului de combustibil și consumul mediu de combustibil (în litri la suta de kilometri).

Prima versiune a clasei **Vehicle** este prezentată mai jos. Aceasta definește trei variabile instanță: **pasageri**, **capcomb** și **cls**. Remarcați că deocamdată clasa **Vehicle** nu conține nici o metodă. În momentul curent, aceasta este deci o clasă care cuprinde numai date. (Vom adăuga metode acestei clase în secțiunile următoare.)

```
class Vehicle {
   public int pasageri; // numarul de pasageri
   public int capcomb;  // capacitatea rezervorului
   public int cls;      // consumul de combustibil, in litri la 100 km
}
```

Variabilele instanță declarate în clasa **Vehicle** exemplifică modul în care se declară variabilele instanță în general. Forma generală a declarației unei variabile instanță este prezentată mai jos:

acces tip nume-var;

Aici, *acces* este specificatorul de acces, *tip* precizează tipul variabilei, iar *var-name* este numele variabilei. Lăsând deci la o parte specificatorul de acces, variabilele instanță se declară în același mod în care se declară și variabilele locale. În clasa

Vehicle, variabilele sunt precedate de modificatorul de acces **public**. După cum s-a explicat, aceasta le permite să fie accesibile pentru codul din afara clasei **Vehicle**.

Definiția unei clase creează un nou tip de date. În acest caz, noul tip de date se numește **Vehicle**. Veți utiliza acest nume pentru a declara obiecte de tipul **Vehicle**. Rețineți însă că o declarație **class** este numai o descriere de tip; ea nu creează efectiv obiecte. Secvența anterioară de cod nu determină deci apariția vreunui obiect de tipul **Vehicle**.

Pentru a crea efectiv un obiect **Vehicle**, trebuie să utilizați o instrucțiune similară cu următoarea:

```
Vehicle minivan = new Vehicle(); // creeaza un obiect Vehicle, numit minivan
```

După execuția acestei instrucțiuni, **minivan** va fi o instanță a clasei **Vehicle**. Prin aceasta, clasa va căpăta o existență „materială“. Pentru moment, nu este cazul să vă preocupați de detaliile acestei instrucțiuni.

De fiecare dată când creați o instanță a unei clase, se creează un obiect care conține o copie proprie a fiecărei variabile instanță definite în cadrul clasei. Prin urmare, orice obiect **Vehicle** va conține cópii proprii ale variabilelor instanță **pasageri**, **capcomb** și **cls**. Pentru a referi aceste variabile, veți utiliza operatorul *punct* (.). Operatorul punct leagă numele unui obiect de numele unui membru al său. Formatul general al operatorului punct este prezentat mai jos:

obiect.membru

În concluzie, obiectul este precizat în stânga operatorului, în timp ce membrul se găsește în partea dreaptă. De exemplu, pentru a atribui variabilei **capcomb** a instanței **minivan** valoarea 80, utilizați următoarea instrucțiune:

```
minivan.capcomb = 80;
```

În general, operatorul punct se utilizează atât pentru accesul la variabilele instanță, cât și la metode.

Iată programul complet care utilizează clasa **Vehicle**:

```
/* Un program care utilizeaza clasa Vehicle.

   Denumiti fisierul UseVehicle.cs
*/

using System;

class Vehicle {
  public int pasageri;    // nr. de pasageri
  public int capcomb;     // capacitatea rezervorului
  public int cls;         // consumul de combustibil, in litri la 100 km
}

// Aceasta clasa declara un obiect de tipul Vehicle.
class VehicleDemo {
  public static void Main() {
```



```
    Vehicle minivan = new Vehicle();
    int dmax;

    // atribuim valori campurilor instantei minivan
    minivan.pasageri = 7;
    minivan.capcomb = 80;
    minivan.cls = 15;

    // calculeaza autonomia de parcurs, presupunand rezervorul initial plin
    dmax = (minivan.capcomb * 100) / minivan.cls;

    Console.WriteLine("Furgoneta poate transporta " + minivan.pasageri + "
                      pasageri pe distanta de " + dmax + " km.");
  }
}
```

Se creează o instanță a lui **Vehicle** numită **minivan**.

Remarcați utilizarea operatorului punct pentru accesul la membri.

Programul constă din două clase: **Vehicle** și **VehicleDemo**. În clasa **VehicleDemo**, metoda **Main()** creează o instanță a clasei **Vehicle** numită **minivan**. Codul din metoda **Main()** accesează apoi variabilele instanță asociate cu **minivan**, atribuindu-le valori și utilizând acele valori. Este foarte important să înțelegeți că **Vehicle** și **VehicleDemo** reprezintă două clase separate. Singura legătură dintre ele este că una din clase creează o instanță a celeilalte. Chiar dacă sunt clase separate, codul din clasa **VehicleDemo** poate accesa membrii clasei **Vehicle**, deoarece aceștia sunt declarați ca **public**. Dacă aceștia nu aveau specificatorul de acces **public**, accesul la ei ar fi fost limitat la clasa **Vehicle**, iar **VehicleDemo** nu ar mai fi putut să-i utilizeze.

Presupunând că ați denumit fișierul precedent **UseVehicle.cs**, compilarea programului va genera un fișier numit **UseVehicle.exe**. Ambele clase **Vehicle** și **VehicleDemo** fac implicit parte din acest fișier executabil. Programul afișează următorul rezultat:

```
Furgoneta poate transporta 7 pasageri pe distanta de 533 km.
```

Nu este obligatoriu ca ambele clase **Vehicle** și **VehicleDemo** să se afle în același fișier sursă. Puteți să includeți ambele clase în fișiere separate, numite de exemplu **Vehicle.cs** și **VehicleDemo.cs**. Trebuie însă în acest caz să cereți compilatorului C# să compileze ambele fișiere, după care să le lege împreună. De exemplu, dacă ați despărțit programul în două fișiere cu numele amintite mai sus, compilați programul utilizând linia de comandă:

```
csc Vehicle.cs VehicleDemo.cs
```

Dacă sunteți în mediul integrat Visual C++, trebuie să adăugați ambele fișiere la proiect și apoi să construiți programul din meniul Build.

Înainte de a trece mai departe, să reținem un principiu fundamental: fiecare obiect deține propriile cópii ale variabilelor instanță definite de clasa sa. Conținutul variabilelor unui obiect diferă în general de conținutul acelorași variabile într-un alt obiect. Nu există nici o legătură între două obiecte, cu excepția faptului că ele sunt de același tip. De exemplu, dacă avem două obiecte **Vehicle**, ambele dețin cópii proprii ale

variabilelor **pasageri**, **capcomb** și **cls**, conținutul acestora fiind de regulă diferit. Programul următor demonstrează această afirmație:

```
// Acest program creeaza doua obiecte Vehicle.

using System;

class Vehicle {
  public int pasageri;   // nr. de pasageri
  public int capcomb;    // capacitatea rezervorului
  public int cls;        // consumul de combustibil, in litri la 100 km
}

// Aceasta clasa declara doua obiecte de tipul Vehicle.
class TwoVehicles {
  public static void Main() {
    Vehicle minivan = new Vehicle();
    Vehicle sportscar = new Vehicle();

    int dmax1, dmax2;

    // atribuim valori campurilor lui minivan
    minivan.pasageri = 7;
    minivan.capcomb = 80;
    minivan.cls = 15;

    // se atribuie valori campurilor instantei sportscar
    sportscar.pasageri = 2;
    sportscar.capcomb = 70;
    sportscar.cls = 20;

    // calculam autonomia de parcurs a celor doua vehicule,
    // presupunand rezervorul initial plin
    dmax1 = (minivan.capcomb * 100) / minivan.cls;
    dmax2 = (sportscar.capcomb * 100) / sportscar.cls;

    Console.WriteLine("Furgoneta poate transporta " + minivan.pasageri +
                      " pasageri pe distanta de " + dmax1 + " km.");

    Console.WriteLine("Masina sport poate transporta " + sportscar.pasageri +
                      " pasageri pe distanta de " + dmax2 + " km.");
  }
}
```

Rețineți că **minivan** și **sportscar** referă obiecte separate.

Rezultatul afișat de program este prezentat mai jos:

```
Furgoneta poate transporta 7 pasageri pe distanta de 533 km.
Masina sport poate transporta 2 pasageri pe distanta de 350 km.
```

După cum se observă, datele instanței **minivan** sunt complet separate de cele conținute de **sportscar**. Situația este reprezentată grafic în figura 4-1.

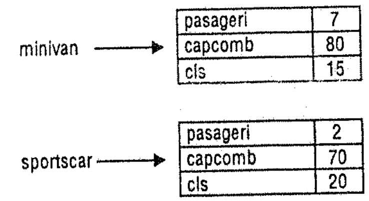


Figura 4-1 Variabilele instanță ale unui obiect sunt separate de cele ale altui obiect.

Exerciții la minut

- Care sunt cele două categorii de entități pe care le conține o clasă?
- Ce operator se utilizează pentru accesul la membrii unei clase prin intermediul unui obiect?
- Fiecare obiect deține propriile cópii ale ____________ ale clasei.

Crearea obiectelor

În programele anterioare, am utilizat linia următoare pentru a declara un obiect de tipul **Vehicle**:

```
Vehicle minivan = new Vehicle();
```

Această instrucțiune are un rol dublu. Mai întâi, ea declară o variabilă numită **minivan**, având ca tip clasa **Vehicle**. Această variabilă nu definește un obiect. Ea este doar o variabilă care poate *referi* un obiect. În al doilea rând, declarația creează o imagine fizică a unui obiect, atribuind variabilei **minivan** o referință către acel obiect. Aceasta se realizează utilizând operatorul **new**. În concluzie, după execuția acestei linii, **minivan** referă un obiect de tipul **Vehicle**.

Operatorul **new** *alocă dinamic* (adică la execuția programului) memorie pentru un obiect și întoarce o referință către acel obiect. Referința reprezintă, mai mult sau mai puțin, adresa în memorie a obiectului alocat de către **new**. Această referință este apoi memorată într-o variabilă. Așadar, în C#, toate obiectele având ca tip clase trebuie alocate dinamic.

Cei doi pași combinați în instrucțiunea anterioară se pot rescrie astfel pentru a evidenția fiecare pas în mod individual:

```
Vehicle minivan;  // declararea referintei la un obiect
minivan = new Vehicle(); // alocarea unui obiect Vehicle
```

- Orice clasă conține cod și date.
- Operatorul folosit pentru accesul la membrii unei clase prin intermediul obiectelor este operatorul punct.
- Fiecare obiect deține propriile cópii ale *variabilelor instanță* ale clasei.

Prima linie declară **minivan** ca referință către un obiect de tipul **Vehicle**. În consecință, **minivan** este o variabilă care poate referi un obiect, dar nu este ea însăși un obiect. În acest moment, **minivan** conține valoarea **null**, ceea ce înseamnă că nu referă nici un obiect. Linia următoare creează un nou obiect de tipul **Vehicle**, atribuind lui **minivan** o referință către acesta. Acum, variabila **minivan** este legată de un obiect.

Faptul că obiectele claselor sunt accesate prin intermediul referințelor justifică denumirea de *tipuri referință* dată claselor. Diferența fundamentală dintre tipurile valorice și tipurile referință apare la semnificația variabilelor aparținând fiecărei categorii de tipuri. La variabilele de tipuri valorice, acestea conțin ele însele o valoare. De exemplu, dacă se dă:

```
int x;
x = 10;
```

Variabila **x** conține valoarea 10, deoarece este de tipul **int**, care este un tip valoric. În cazul instrucțiunii:

```
Vehicle minivan = new Vehicle();
```

variabila **minivan** nu conține ea însăși un obiect, ci o referință către un obiect.

Atribuirea valorilor pentru variabilele referință

Într-o operație de atribuire, variabilele referință se comportă diferit față de cele de tipuri valorice, cum este **int**. La atribuirea unei variabile de tip valoric altei variabile, situația este foarte clară. Variabila din partea stângă primește o copie a valorii variabilei din dreapta. Atunci când însă atribuiți o variabilă care este referința unui obiect alteia, situația este ceva mai complicată, deoarece se schimbă obiectul pe care variabila referință îl referă. Efectul acestei diferențe poate determina câteva rezultate nu tocmai conforme cu așteptările. De exemplu, să considerăm următorul fragment:

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
```

La o primă privire, pare evident că variabilele **car1** și **car2** referă obiecte diferite, dar lucrurile nu stau așa. Cele două variabile **car1** și **car2** referă ambele *același* obiect. Atribuirea lui **car1** către **car2** determină ca și **car2** să refere același obiect cu **car1**. Puteți deci acționa asupra obiectului utilizând fie **car1**, fie **car2**. De exemplu, după executarea atribuirii:

```
car1.cls = 12;
```

ambele instrucțiuni **WriteLine()** de mai jos vor afișa aceeași valoare, 12:

```
Console.WriteLine(car1.cls);
Console.WriteLine(car2.cls);
```


Deși **car1** și **car2** referă ambele același obiect, între ele nu există nici o altă legătură. De exemplu, o atribuire ulterioară a lui **car2** poate modifica obiectul pe care **car2** îl referă. De exemplu:

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
Vehicle car3 = new Vehicle();

car2 = car3;   // acum, car2 si car3 refera acelasi obiect.
```

După execuția acestei secvențe, **car2** va referi același obiect ca și **car3**. Obiectul referit de **car1** rămâne nemodificat.

Exerciții la minut

- Explicați ce se întâmplă atunci când o variabilă referință este atribuită alteia.
- Presupunând că aveți o clasă numită **MyClass**, arătați cum puteți crea un obiect **ob** aparținând acestei clase.
- Care este diferența dintre tipurile valorice și tipurile referință?

Metode

După cum am precizat, variabilele instanță și metodele sunt două dintre componentele primare ale claselor. Până acum, clasa **Vehicle** conține date, dar nu și metode. Deși clasele care conțin numai date sunt perfect corecte, majoritatea claselor au metode. Metodele sunt subrutine care prelucrează datele definite în cadrul clasei, iar, în multe cazuri, oferă accesul la acele date. De regulă, interacțiunea dintre o clasă și celelalte entități din program se face prin intermediul metodelor clasei.

O metodă poate conține una sau mai multe instrucțiuni. În programele C# scrise corect, fiecare metodă îndeplinește o singură acțiune. Fiecare metodă are un nume, acesta fiind folosit pentru a apela metoda. În general, puteți da unei metode orice nume doriți. Amintiți-vă însă că numele **Main()** este rezervat pentru metoda cu care începe execuția programului. De asemenea, nu utilizați cuvinte cheie din C# ca nume de metode.

La indicarea metodelor în text, această carte a folosit până acum și va continua să folosească o convenție care a devenit uzuală în literatura de specialitate. Numele metodelor vor fi urmate de paranteze. De exemplu, dacă numele unei metode este **getval**,

---

- La atribuirea unei variabile referință alteia, ambele variabile vor referi același obiect. Nu se face o copie a obiectului.
- `MyClass ob = new MyClass();`
- Variabilele de tipuri valorice conțin ele însele valori. Cele de tip referință referă un alt obiect.

acesta se va scrie **getval()** atunci când va fi utilizat într-o propoziție. Această notație vă va ajuta să deosebiți numele variabilelor de numele metodelor din această carte.

Forma generală a unei metode este prezentată mai jos:

acces tip-rez nume(lista-parametri) {
    // corpul metodei
}

Aici, *acces* este un modificator de acces care stabilește care dintre celelalte părți ale programului poate apela metoda. După cum am explicat mai devreme, modificatorul de acces este opțional. Dacă acesta lipsește, metoda este privată clasei în care a fost declarată. Pentru început, vom declara metodele ca **public**, pentru a putea fi apelate de oriunde din program. Tipul rezultatului pe care metoda îl întoarce este specificat de către *tip-rez*. Acesta poate fi orice tip valid, inclusiv unul din tipurile clasă create de dumneavoastră. Dacă metoda nu întoarce nici o valoare, tipul rezultatului său trebuie să fie **void**. Numele metodei este specificat prin *nume*. Acesta poate fi orice identificator corect, diferit de cele deja utilizate de alte entități din domeniul său de valabilitate. *Lista-parametri* este o secvență de perechi în care primul element este un tip, iar al doilea un identificator; aceste perechi sunt separate prin virgulă. Parametrii sunt de fapt variabile care primesc valorile efective transmise metodei atunci când este apelată. Dacă metoda nu are parametri, lista de parametri este vidă.

Adăugarea unei metode la clasa `Vehicle`

După cum tocmai am precizat, metodele unei clase prelucrează și asigură accesul la datele clasei. Reținând acest aspect, să ne amintim că metoda **Main()** din exemplele anterioare calcula autonomia de parcurs a unui vehicul împărțind capacitatea rezervorului la consumul pe 100 km. Chiar dacă din punct de vedere tehnic este corect, nu este cea mai eficientă modalitate de rezolvare a problemei. Calculul autonomiei de parcurs a unui vehicul reprezintă o acțiune care este cel mai bine să fie rezolvată în cadrul clasei **Vehicle** însăși. Motivul acestei concluzii este lesne de înțeles: autonomia unui vehicul depinde de capacitatea rezervorului de benzină și de consumul mediu al vehiculului, ambele informații fiind încapsulate în clasa **Vehicle**. Adăugând clasei **Vehicle** o metodă care determină autonomia de parcurs, puteți îmbunătăți structura orientată spre obiecte a clasei.

Pentru adăugarea unei metode la clasa **Vehicle**, aceasta trebuie specificată în declarația clasei. De exemplu, următoarea versiune a clasei **Vehicle** conține o metodă numită **aut()**, care afișează autonomia de parcurs a vehiculului:

```
// Adaugarea autonomiei de parcurs.

using System;
```



```
class Vehicle {
  public int pasageri;   // nr. de pasageri
  public int capcomb;    // capacitatea rezervorului
  public int cls;        // consumul de combustibil, in litri la 100 km

  // Afiseaza autonomia.
  public void aut() {
    Console.WriteLine("Autonomia este de " + 100 * capcomb / cls + " km.");
  }
}

class AddMeth {
  public static void Main() {
    Vehicle minivan = new Vehicle();
    Vehicle sportscar = new Vehicle();

    // atribuim valori campurilor lui minivan
    minivan.pasageri = 7;
    minivan.capcomb = 80;
    minivan.cls = 15;

    // se atribuie valori campurilor instantei sportscar
    sportscar.pasageri = 2;
    sportscar.capcomb = 70;
    sportscar.cls = 20;

    Console.Write("Furgoneta poate transporta " + minivan.pasageri +
                  " pasageri.");

    minivan.aut();   // afisarea autonomiei lui minivan

    Console.Write("Masina sport poate transporta " + sportscar.pasageri +
                  " pasageri.");

    sportscar.aut();   // afisarea autonomiei lui minivan
  }
}
```

Metoda **aut()** aparține clasei **Vehicle**.

Remarcați utilizarea directă a lui **capcomb** și **cls**, fără operatorul punct.

Programul va afișa următorul rezultat:

```
Furgoneta poate transporta 7 pasageri. Autonomia este de 533 km.
Masina sport poate transporta 2 pasageri. Autonomia este de 350 km.
```

Să examinăm elementele fundamentale ale programului, începând cu metoda **aut()** însăși. Prima linie a metodei **aut()** este:

```
public void aut() {
```

Aceasta declară o metodă numită **aut**, fără parametri. Metoda este specificată ca **public**, deci va putea fi apelată din orice altă parte a programului. Tipul rezultatului său este **void**. În concluzie, **aut()** nu întoarce nici o valoare apelantului său. Linia se încheie cu acolada de deschidere a corpului metodei.

Corpul metodei **aut()** constă numai din linia:

```
Console.WriteLine("Autonomia este de " + 100 * capcomb / cls + " km.");
```

Această instrucțiune afișează autonomia de parcurs a unui vehicul împărțind variabila instanță **capcomb**, înmulțită cu 100, la **cls**. Deoarece fiecare obiect de tipul **Vehicle** dispune de o copie proprie a variabilelor **capcomb** și **cls**, la apelul metodei **aut()**, calculul autonomiei va utiliza copiile variabilelor din obiectul apelant.

Metoda **aut()** se termină la întâlnirea acoladei sale de închidere. Aceasta determină transferul controlului înapoi către modulul apelant.

În continuare, să studiem mai amănunțit această linie din metoda **Main()**:

```
minivan.aut();
```

Această instrucțiune apelează metoda **aut()** asupra lui **minivan**. Cu alte cuvinte, se apelează **aut()** relativ la obiectul **minivan**, utilizând numele obiectului, urmat de operatorul punct. La apelul unei metode, controlul programului este transferat acelei metode. Atunci când metoda se termină, controlul este transferat înapoi modulului apelant, iar execuția continuă cu linia imediat următoare apelului.

În acest caz, apelul **minivan.aut()** afișează autonomia de parcurs a vehiculului definit de **minivan**. În mod similar, apelul **sportscar.aut()** afișează autonomia vehiculului definit de către **sportscar**. De fiecare dată când este invocată, metoda **aut()** afișează autonomia de parcurs a obiectului specificat.

Există un amănunt foarte important de remarcat în corpul metodei **aut()**: variabilele instanță **capcomb** și **cls** sunt referite direct, fără a fi precedate de numele vreunui obiect sau de operatorul punct. Atunci când o metodă utilizează o variabilă instanță definită în aceeași clasă, o face în mod direct, fără vreo referință explicită către un obiect și fără să recurgă la operatorul punct. Este simplu de înțeles de ce se întâmplă așa dacă ne gândim puțin. O metodă este întotdeauna invocată relativ la un obiect din clasa sa. După ce invocarea a avut loc, obiectul devine cunoscut. În cadrul unei metode, nu mai este deci nevoie să specificăm obiectul a doua oară. Aceasta înseamnă că identificatorii **capcomb** și **cls** din metoda **aut()** referă în mod implicit cópiile variabilelor instanță respective din obiectul care a invocat **aut()**.

Revenirea dintr-o metodă

În general, există două condiții care determină revenirea dintr-o metodă. Prima, cum este cazul și metodei **aut()** din exemplul precedent, se produce când se întâlnește acolada de închidere a metodei. Cea de-a doua presupune execuția unei instrucțiuni **return**. Există două forme ale instrucțiunii **return**: una care se utilizează în metode **void** (cele care nu întorc nici o valoare) și cealaltă care întoarce valori. Vom începe examinând prima formă. Secțiunea următoare prezintă modalitatea în care puteți întoarce valori.

Terminarea imediată a unei metode **void** se produce dacă se utilizează forma următoare a lui **return**:

```
return;
```


La execuția acestei instrucțiuni, controlul programului revine în modulul apelant, ignorând codul rămas în corpul metodei apelate. Spre exemplu, să considerăm metoda:

```
public void MyMeth() {
  int i;

  for(i = 0; i<10; i++) {
    if(i == 5) return;    // oprire la 5
    Console.WriteLine();
  }
}
```

Aici, bucla **for** va itera numai de la 0 la 5, deoarece atunci când **i** devine 5, metoda va reveni.

Se admite ca într-o metodă să existe mai multe instrucțiuni **return**, în special atunci când există cel puțin două ramuri de ieșire din metodă. De exemplu,

```
public void myMeth() {
  // ...
  if(terminat) return;
  // ...
  if(eroare) return;
}
```

În acest caz, metoda revine dacă a terminat o anumită prelucrare sau dacă apare o eroare. Se impune însă să fiți precaut. Prea multe puncte de ieșire dintr-o metodă înrăutățesc structurarea codului, deci evitați utilizarea lor neglijentă.

Să recapitulăm: o metodă **void** poate reveni în două moduri: la întâlnirea acoladei de închidere, sau prin execuția unei instrucțiuni **return**.

Întoarcerea unei valori

Chiar dacă metodele care întorc un rezultat de tip **void** sunt destul de frecvent întâlnite, majoritatea metodelor întorc totuși o valoare. De fapt, posibilitatea de a întoarce o valoare constituie una dintre cele mai utile facilități ale unei metode. Ați văzut deja un exemplu de metodă care întoarce o valoare atunci când am utilizat funcția **Math.Sqrt()** pentru a calcula rădăcina pătrată a unui număr.

Valorile întoarse se utilizează într-o mare diversitate de scopuri în programare. În unele cazuri, cum se întâmplă și cu **Math.Sqrt()**, valoarea întoarsă reprezintă rezultatul unui calcul. În alte cazuri, valoarea întoarsă indică pur și simplu reușita sau eșecul unei operații. În altele, poate conține un cod de stare. Indiferent de scopul urmărit, utilizarea valorilor întoarse de către metode reprezintă o parte esențială a programării în C#.

Metodele pot întoarce o valoare în rutina apelantă utilizând următoarea formă a instrucțiunii **return**:

return *val*;

Aici, *val* reprezintă valoarea întoarsă.

Puteți îmbunătăți implementarea metodei **aut()** prevăzând ca metoda să întoarcă o valoare. În loc de a afișa autonomia de parcurs, o soluție mai bună este ca **aut()** să calculeze valoarea autonomiei, întorcând-o ca rezultat. Unul dintre avantajele acestei soluții este că puteți utiliza mai departe această valoare în alte calcule. Exemplul următor modifică metoda **aut()** astfel încât să întoarcă valoarea autonomiei de parcurs în loc de a o afișa:

```
// Utilizarea unei valori de intoarcere.

using System;

class Vehicle {
  public int pasageri;   // nr. de pasageri
  public int capcomb;    // capacitatea rezervorului
  public int cls;        // consumul de combustibil, in litri la 100 km

  // Intoarce autonomia.
  public int aut() {
    return (100 * capcomb / cls);
  }
}

  class RetMeth {
    public static void Main() {
      Vehicle minivan = new Vehicle();
      Vehicle sportscar = new Vehicle();

      int aut1, aut2;

    // atribuim valori campurilor lui minivan
    minivan.pasageri = 7;
    minivan.capcomb = 80;
    minivan.cls = 15;

    // se atribuie valori campurilor instantei sportscar
    sportscar.pasageri = 2;
    sportscar.capcomb = 70;
    sportscar.cls = 20;

    // calculam autonomiile
    aut1 = minivan.aut();
    aut2 = sportscar.aut();

    Console.WriteLine("Furgoneta poate transporta " + minivan.pasageri +
                      " pasageri cu o autonomie de " + aut1 + " km.");

    Console.WriteLine("Masina sport poate transporta " + sportscar.pasageri +
                      " pasageri cu o autonomie de " + aut2 + " km.");
  }
}
```

aut() întoarce acum o valoare.

Se atribuie valoarea întoarsă unei variabile.

Rezultatul afișat este prezentat mai jos:

```
Furgoneta poate transporta 7 pasageri cu o autonomie de 533 km.
Masina sport poate transporta 2 pasageri cu o autonomie de 350 km.
```


În program, remarcați că apelul metodei **aut()** apare în partea dreaptă a unei instrucțiuni de atribuire. În partea stângă se găsește o variabilă care primește valoarea întoarsă de **aut()**. Astfel, după execuția liniei următoare,

```
aut1 = minivan.aut();
```

autonomia de parcurs a obiectului **minivan** este memorată în **aut1**.

Observați că metoda **aut()** întoarce acum un rezultat de tipul **int**. Aceasta înseamnă că metoda întoarce în modulul apelant o valoare întreagă. Tipul rezultatului unei metode este foarte important, deoarece tipul datelor pe care metoda le întoarce trebuie să fie compatibil cu tipul specificat de antetul metodei. În consecință, dacă doriți ca o metodă să întoarcă o dată de tipul **double**, tipul rezultatului specificat în antetul metodei trebuie să fie tot **double**.

Deși programul anterior este corect, nu este scris pe cât de eficient este posibil. Mai precis, variabilele **aut1** și **aut2** nu sunt necesare. Se poate utiliza apelul metodei **aut()** direct în instrucțiunea **WriteLine()**, după cum se observă mai jos:

```
Console.WriteLine("Furgoneta poate transporta " + minivan.pasageri +
                  " pasageri cu o autonomie de " + minivan.aut() + " km.");
```

În acest caz, la execuția instrucțiunii **WriteLine()**, **minivan.aut()** este apelată automat, iar valoarea întoarsă este transmisă lui **WriteLine()**. Mai mult, puteți utiliza apelul metodei **aut()** ori de câte ori este nevoie de calculul autonomiei unui obiect de tipul **Vehicle**. De exemplu, instrucțiunea de mai jos compară autonomiile a două vehicule:

```
if(v1.aut() > v2.aut()) Console.WriteLine("v1 are autonomia mai mare");
```

Sfatul expertului

Întrebare: Am auzit că în C# se detectează „codul inaccesibil". Ce înseamnă de fapt aceasta?

Răspuns: Ați auzit bine. Complatorul de C# generează un mesaj de avertizare dacă ați creat o metodă în care există cod care nu poate fi executat. Să considerăm exemplul:

```
public void m() {
  char a, b;

  // ...

  if(a == b) {
    Console.WriteLine("caractere egale");
    return;
  } else {
    Console.WriteLine("caractere diferite");
    return;
  }
  Console.WriteLine("Instructiune inaccesibila");
}
```

Aici, metoda `m()` va reveni înotdeauna înainte de execuția instrucțiunii `WriteLine()` finale. Dacă încercați să compilați această metodă, compilatorul va afișa un avertisment. În general, codul inaccesibil constituie o eroare din partea programatorului, deci se recomandă să tratați aceste avertismente cu seriozitate.

Utilizarea parametrilor

Este posibil să transmitem una sau mai multe valori unei metode atunci când aceasta este apelată. După cum s-a explicat, o valoare transmisă unei metode este denumită *parametru efectiv* (sau argument). În corpul metodei, variabilele care primesc valorile parametrilor efectivi se numesc *parametri formali*. Parametrii formali sunt declarați în interiorul parantezelor care urmează după numele metodei. Sintaxa declarației parametrilor formali este aceeași ca și în cazul oricăror variabile. Parametrii formali au ca domeniu de valabilitate corpul metodei, iar în afara rolului lor special de a primi valorile parametrilor efectivi, aceștia se comportă exact ca și orice alte variabile locale.

Iată un exemplu simplu de metodă cu un parametru. În interiorul clasei `ChkNum`, metoda `isEven()` întoarce `true` dacă valoarea pe care o primește este pară. În caz contrar, metoda întoarce `false`. În concluzie, metoda întoarce un rezultat de tipul `bool`.

```
// Un exemplu simplu de metoda cu un parametru.

using System;

class ChkNum {
  // intoarce true daca x este par
  public bool isEven(int x) {
    if((x % 2) == 0) return true;
    else return false;
  }
}

class ParmDemo {
  public static void Main() {
    ChkNum e = new ChkNum();

    if(e.isEven(10)) Console.WriteLine("10 este par.");

    if(e.isEven(9)) Console.WriteLine("9 este par.");

    if(e.isEven(8)) Console.WriteLine("8 este par.");
  }
}
```

x este parametrul întreg al lui `isEven()`.

Rezultatul afișat de program este:

```
10 este par.
8 este par.
```


În program, metoda `isEven()` este apelată de trei ori, dar de fiecare dată i se transmite o altă valoare. Să studiem acest proces mai în amănunt. Mai întâi, să observăm modul în care este apelată `isEven()`. Parametrul efectiv este specificat între paranteze. La primul apel, `isEven()` primește valoarea 10. Ca urmare, atunci când metoda `isEven()` începe să se execute, parametrul formal `x` primește valoarea 10. La al doilea apel, parametrul efectiv este 9, deci `x` va primi tot valoarea 9. La al treilea apel, parametrul efectiv este 8, valoare primită tot de `x`. Concluzia este că valoarea transmisă ca parametru efectiv la apelul lui `isEven()` este chiar valoarea pe care o primește parametrul formal al metodei, `x`.

O metodă poate avea mai mulți parametri. Aceștia se declară în mod simplu, separându-se între ei prin virgulă. De exemplu, clasa `Divizor` definește o metodă numită `esteDivizor()` care stabilește dacă primul parametru este un divizor al celui de-al doilea.

```
using System;

class Divizor() {
  public bool esteDivizor(int a, int b) {
    if((b % a) == 0) return true;
    else return false;
  }
}

class EsteDiv {
  public static void Main() {
    Divizor x = new Divizor();

    if(x.esteDivizor(2, 20)) Console.WriteLine("2 este divizorul lui 20");
    if(x.esteDivizor(3, 20)) Console.WriteLine("Nu se va afisa");
  }
}
```

Această metodă are doi parametri.

Remarcați că și la apelul metode `esteDivizor()`, parametrii efectivi sunt separați prin virgulă.

Atunci când sunt mai mulți parametri, fiecare dintre aceștia își specifică tipul, care poate fi diferit de tipul celorlalți. De exemplu, următoarea linie este perfect corectă:

```
int myMeth(int a, double b, float c) {
// ...
```

Adăugarea unei metode parametrizate la clasa `Vehicle`

Puteți acum utiliza o metodă parametrizată pentru a adăuga o nouă facilitate la clasa `Vehicle`: posbilitatea de a calcula cantitatea de combustibil necesară pentru parcurgerea unei distanțe date. Această nouă metodă este denumită `combnec()`. Această metodă primește ca informație distanța pe care doriți să o parcurgeți și întoarce nu-

mărul de litri de benzină necesari pentru aceasta. Metoda **combnec()** este definită după cum urmează:

```
public double combnec(int dist) {
  return (double) cls * dist / 100.0;
}
```

Remarcaţi că metoda întoarce o valoare de tipul **double**. Tipul ales este corect, deoarece cantitatea de combustibil necesară parcurgerii unei distanţe date poate să nu fie un număr întreg.

Codul complet al clasei **Vehicle**, inclusiv noua metodă **combnec()**, este prezentat mai jos:

```
/*
   Adaugarea unei metode parametrizate care calculeaza
   cantitatea de combustibil necesara pe o anumita distanta.
*/

using System;

class Vehicle {
  public int pasageri;    // nr. de pasageri
  public int capcomb;     // capacitatea rezervorului
  public int cls;         // consumul de combustibil, in litri la 100 km

  // Intoarce autonomia.
  public int aut() {
    return (100 * capcomb / cls);
  }

  // Calculeaza combustibilul necesar pe o distanta data.
  public double combnec(int dist) {
    return (double) cls * dist / 100.0;
  }
}

class CalcComb {
  public static void Main() {
    Vehicle minivan = new Vehicle();
    Vehicle sportscar = new Vehicle();
    double consum;
    int dist = 407;

    // atribuim valori campurilor lui minivan
    minivan.pasageri = 7;
    minivan.capcomb = 80;
    minivan.cls = 15;

    // se atribuie valori campurilor instantei sportscar
    sportscar.pasageri = 2;
    sportscar.capcomb = 70;
    sportscar.cls = 20;

    consum = minivan.combnec(dist);
```



```
    Console.WriteLine("Pe distanta de " + dist + " km, furgoneta consuma "
                     + consum + " litri de benzina.");

    consum = sportscar.combnec(dist);

    Console.WriteLine("Pe distanta de " + dist + " km, masina sport consuma "
                     + consum + " litri de benzina.");
  }
}
```

Rezultatul afişat de program este prezentat mai jos:

```
Pe distanta de 407 km, furgoneta consuma 61.05 litri de benzina.
Pe distanta de 407 km, masina sport consuma 81.40 litri de benzina.
```

Exerciţii la minut

- Când trebuie accesate variabilele instanţă sau metodele prin intermediul unei referinţe la un obiect, utilizând operatorul punct? Când pot fi utilizate direct?
- Explicaţi diferenţa dintre parametrii efectivi şi parametrii formali.
- Prezentaţi cele două moduri în care o metodă poate reveni în modulul apelant.

Proiectul 4-1: Crearea unei clase de help

Încercând să sintetizăm esenţa conceptului de clasă într-o singură propoziţie, am putea formula astfel: clasele încapsulează funcţionalitate. Desigur, uneori problema este de a şti unde se termină o anume „funcţionalitate" şi unde începe alta. Ca regulă generală, clasele pe care le veţi construi vor fi piesele din care veţi asambla aplicaţii de dimensiuni mai mari. Pentru aceasta, fiecare clasă trebuie să reprezinte o singură unitate funcţională care execută acţiuni conturate în mod precis. În concluzie, clasele dumneavoastră trebuie să aibă dimensiuni pe cât posibil de reduse – dar nu mai mici decât este necesar. Aşadar, clasele care adaugă funcţionalitate neesenţială complică inutil codul şi îi înrăutăţesc structurarea, dar pe de altă parte clasele cu prea puţină funcţionalitate se prezintă ca fragmentate. Unde este punctul de echilibru? Acesta

---

- Variabilele instanţă pot fi accesate de codul care nu face parte din clasa în care sunt definite numai prin intermediul unui obiect, utilizând operatorul punct. Atunci când variabilele instanţă sunt accesate de cod care face parte din aceeaşi clasă cu ele, pot fi referite în mod direct. Acelaşi lucru este valabil şi pentru metode.
- Un *parametru efectiv* este o valoare transmisă unei metode când acesta este invocată. Un *parametru formal* este o variabilă definită de către o metodă şi care primeşte valoarea unui parametru efectiv.
- Revenirea unei metode se produce atunci când se utilizează instrucţiunea **return**. Dacă tipul rezultatului metodei este **void**, metoda revine şi la întâlnirea acoladei de închidere. Metodele care au tipul diferit de **void** trebuie să întoarcă o valoare, deci nu dispun de opţiunea revenirii la întâlnirea acoladei de închidere.

este punctul în care *știința* programării devine o *artă*. Din fericire, majoritatea programatorilor constată că determinarea acestui echilibru se simplifică pe măsura creșterii experienței.

Pentru a începe să acumulați experiență, veți converti sistemul de help creat în proiectul 3-3 într-o clasă **Help**. Să precizăm ce avantaje ne aduce această idee. În primul rând, sistemul de help definește o singură unitate logică. Acesta afișează pur și simplu sintaxa instrucțiunilor de control din C#. Funcționalitatea sa este deci compactă și bine definită. În al doilea rând, includerea sistemului de help într-o clasă reprezintă o abordare extrem de elegantă. Ori de câte ori doriți să includeți sistemul de help într-o aplicație, este suficient să instanțiați clasa **Help**. În fine, datorită încapsulării, sistemul de help poate fi îmbunătățit sau modificat fără a produce efecte secundare nedorite în programele care îl utilizează.

Pas cu pas

1. Creați un nou fișier, numit **HelpClassDemo.cs**. Pentru a câștiga ceva timp la introducerea programului, puteți copia conținutul fișierului din proiectul 3-3, numit **Help3.cs**, în **HelpClassDemo.cs**.
2. Pentru a converti sistemul de help într-o clasă, trebuie să stabiliți mai întâi cu exactitate componentele care fac parte din sistem. De exemplu, în fișierul **Help3.cs**, există cod care afișează un meniu, citește opțiunea utilizatorului, verifică dacă răspunsul este valid și afișează informația referitoare la opțiunea selectată. Programul rămâne într-o buclă până la apăsarea tastei *q*. Dacă vă gândiți mai bine, este evident că meniul, verificarea corectitudinii răspunsului și afișarea informației de documentare sunt parte integrantă a sistemului de help. Modul de obținere a opțiunii utilizatorului și prelucrarea cererilor repetate nu fac însă parte din sistem. Veți crea deci o clasă care afișează informația de documentare și meniul, verificând corectitudinea opțiunii selectate. Metodele corespunzătoare acestor acțiuni se vor numi **helpon()**, **showmenu()** și respectiv **isvalid()**.
3. Creați metoda **helpon()** după cum observați în continuare:

```
public void helpon(char what) {
  switch(what) {
  case '1':
    Console.WriteLine("Instructiunea if:\n");
    Console.WriteLine("if(conditie) instructiune;");
    Console.WriteLine("else instructiune;");
    break;
  case '2':
    Console.WriteLine("Instructiunea switch:\n");
    Console.WriteLine("switch(expresie) {");
    Console.WriteLine("  case constanta:");
    Console.WriteLine("    secventa de instructiuni");
    Console.WriteLine("    break;");
    Console.WriteLine("  // ...");
    Console.WriteLine("}");
    break;
  case '3':
    Console.WriteLine("Instructiunea for:\n");
    Console.WriteLine("for(init; conditie; iteratie)");
    Console.WriteLine(" instructiune;");
    break;
  case '4':
    Console.WriteLine("Instructiunea while:\n");
    Console.WriteLine("while(conditie) instructiune;");
  break;
```



```
case '5':
    Console.WriteLine("Instructiunea do-while:\n");
    Console.WriteLine("do {");
    Console.WriteLine(" instructiune;");
    Console.WriteLine("} while(conditie);");
    break;
  case '6':
    Console.WriteLine("Instructiunea break:\n");
    Console.WriteLine("break;");
    break;
  case '7':
    Console.WriteLine("Instructiunea continue:\n");
    Console.WriteLine("continue;");
    break;
  case '8':
    Console.WriteLine("Instructiunea goto:\n");
    Console.WriteLine("goto eticheta;");
    break;
  }
  Console.WriteLine();
}
```

4. Creați acum metoda **showmenu()**:

```
public void showmenu() {
  Console.WriteLine("Help la:");
  Console.WriteLine("  1. if");
  Console.WriteLine("  2. switch");
  Console.WriteLine("  3. for");
  Console.WriteLine("  4. while");
  Console.WriteLine("  5. do-while\n");
  Console.WriteLine("  6. break\n");
  Console.WriteLine("  7. continue\n");
  Console.WriteLine("  8. goto\n");
  Console.WriteLine("Alegerea dumneavoastra (q pentru terminare): ");
}
```

5. Creați metoda **isvalid()**, prezentată mai jos:

```
public bool isvalid(char ch) {
  if(ch < '1' | ch > '8' & ch != 'q') return false;
  else return true;
}
```

6. Includeți acum cele trei metode de mai sus în cadrul clasei **Help**, prezentată complet mai jos:

```
class Help {
  public void helpon(char what) {
    switch(what) {
      case '1':
        Console.WriteLine("Instructiunea if:\n");
        Console.WriteLine("if(conditie) instructiune;");
        Console.WriteLine("else instructiune;");
      break;
      case '2':
        Console.WriteLine("Instructiunea switch:\n");
        Console.WriteLine("switch(expresie) {");
        Console.WriteLine("  case constanta:");
        Console.WriteLine("    secventa de instructiuni");
        Console.WriteLine("    break;");
```

```
      Console.WriteLine("    // ...");
      Console.WriteLine("}");
      break;
    case '3':
      Console.WriteLine("Instructiunea for:\n");
      Console.WriteLine("for(init; conditie; iteratie)");
      Console.WriteLine(" instructiune;");
      break;
    case '4':
      Console.WriteLine("Instructiunea while:\n");
      Console.WriteLine("while(conditie) instructiune;");
      break;
    case '5':
      Console.WriteLine("Instructiunea do-while:\n");
      Console.WriteLine("do {");
      Console.WriteLine(" instructiune;");
      Console.WriteLine("} while(conditie);");
      break;
    case '6':
      Console.WriteLine("Instructiunea break:\n");
      Console.WriteLine("break;");
      break;
    case '7':
      Console.WriteLine("Instructiunea continue:\n");
      Console.WriteLine("continue;");
      break;
    case '8':
      Console.WriteLine("Instructiunea goto:\n");
      Console.WriteLine("goto eticheta;");
      break;
  }
  Console.WriteLine();
}

public void showmenu() {
  Console.WriteLine("Help la:");
  Console.WriteLine("   1. if");
  Console.WriteLine("   2. switch");
  Console.WriteLine("   3. for");
  Console.WriteLine("   4. while");
  Console.WriteLine("   5. do-while\n");
  Console.WriteLine("   6. break\n");
  Console.WriteLine("   7. continue\n");
  Console.WriteLine("   8. goto\n");
  Console.WriteLine("Alegerea dumneavoastra (q pentru terminare): ");
}

public bool isvalid(char ch) {
  if(ch < '1' | ch > '8' & ch != 'q') return false;
  else return true;
}
}
```

7. În fine, rescrieți metoda **Main()** din proiectul 3-3 astfel încât să poată utiliza noua clasă **Help**. Denumiți clasa care conține **Main() HelpClassDemo**. Listingul complet al programului **HelpClassDemo.cs** este prezentat în cele ce urmează:


```
/*
  Proiectul 4-1

  Transformarea sistemului de help din proiectul 3-3 intr-o clasa Help.
*/

using System;

class Help {
  public void helpon(char what) {
    switch(what) {
      case '1':
        Console.WriteLine("Instructiunea if:\n");
        Console.WriteLine("if(conditie) instructiune;");
        Console.WriteLine("else instructiune;");
        break;
      case '2':
        Console.WriteLine("Instructiunea switch:\n");
        Console.WriteLine("switch(expresie) {");
        Console.WriteLine("   case constanta:");
        Console.WriteLine("secventa de instructiuni");
        Console.WriteLine(" break;");
        Console.WriteLine("    // ...");
        Console.WriteLine("}");
        break;
      case '3':
        Console.WriteLine("Instructiunea for:\n");
        Console.WriteLine("for(init; conditie; iteratie)");
        Console.WriteLine(" instructiune;");
        break;
      case '4':
        Console.WriteLine("Instructiunea while:\n");
        Console.WriteLine("while(conditie) instructiune;");
        break;
      case '5':
        Console.WriteLine("Instructiunea do-while:\n");
        Console.WriteLine("do {");
        Console.WriteLine(" instructiune;");
        Console.WriteLine("} while(conditie);");
        break;
      case '6':
        Console.WriteLine("Instructiunea break:\n");
        Console.WriteLine("break;");
        break;
      case '7':
        Console.WriteLine("Instructiunea continue:\n");
        Console.WriteLine("continue;");
        break;
      case '8':
        Console.WriteLine("Instructiunea goto:\n");
        Console.WriteLine("goto eticheta;");
        break;
    }
    Console.WriteLine();
  }
```

```
  public void showmenu() {
    Console.WriteLine("Help la:");
    Console.WriteLine("  1. if");
    Console.WriteLine("  2. switch");
    Console.WriteLine("  3. for");
    Console.WriteLine("  4. while");
    Console.WriteLine("  5. do-while\n");
    Console.WriteLine("  6. break\n");
    Console.WriteLine("  7. continue\n");
    Console.WriteLine("  8. goto\n");
    Console.WriteLine("Alegerea dumneavoastra (q pentru terminare): ");
  }

  public bool isvalid(char ch) {
    if(ch < '1' | ch > '8' & ch != 'q') return false;
    else return true;
  }
}

class HelpClassDemo {
  public static void Main() {
    char choice;
    Help hlpobj = new Help();

    for(; ;) {
      do {
        hlpobj.showmenu();
        do {
          choice = (char) Console.Read();
        } while(choice == '\n' | choice == '\r');

      } while( ! hlpobj.isvalid(choice));

      if(choice == 'q') break;

      Console.WriteLine("\n");

      hlpobj.helpon(choice);
    }
  }
}
```

Încercând să executați programul, veți observa că, din punct de vedere funcțional, nu s-a modificat față de versiunea anterioară. Avantajul acestei abordări este că acum puteți reutiliza componenta „sistem de help" ori de câte ori aveți nevoie.

Constructori

În exemplele anterioare, variabilele instanță din fiecare obiect de tipul **Vehicle** erau inițializate manual, utilizând o secvență de instrucțiuni de forma:

```
minivan.pasageri = 7;
minivan.capcomb = 80;
minivan.cls = 15;
```


O soluție de acest gen nu o veți întâlni niciodată în cadrul programelor C# „de firmă". Pe lângă predispoziția la erori (este posibil să omiteți inițializarea unui câmp), motivul este că există o metodă mult mai bună de a realiza această operație: utilizarea unui constructor.

Un *constructor* inițializează un obiect atunci când acesta este creat. Constructorii au același nume cu clasa din care fac parte, fiind similari din punct de vedere sintactic cu metodele obișnuite. Constructorii nu au însă un tip explicit. Forma generală a unui constructor este prezentată mai jos:

nume-clasa () {
  // codul constructorului
}

De regulă, constructorii se utilizează pentru atribuirea valorilor inițiale pentru variabilele instanță definite în cadrul clasei, sau pentru efectuarea altor operații inițiale necesare la crearea unui obiect complet inițializat.

Toate clasele dispun de constructori, indiferent dacă îi definiți sau nu, deoarece limbajul C# pune la dispoziție un constructor implicit, care inițializează toate variabilele membru cu zero (pentru tipuri valorice), respectiv cu null (pentru tipuri referință). Dacă ați definit însă propriul dumneavoastră constructor, cel implicit nu va mai fi utilizat.

Mai jos este prezentat un exemplu simplu care utilizează un constructor:

```
// Un constructor simplu.

using System;

class MyClass {
  public int x;

  public MyClass() {   <-- Constructorul clasei MyClass
    x = 10;
  }
}

class ConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass();

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

În acest exemplu, constructorul clasei **MyClass** este

```
public MyClass() {
  x = 10;
}
```

Observați că acesta este specificat ca fiind **public**. Aceasta se întâmplă deoarece constructorul va fi apelat de cod definit în afara clasei din care face parte.

Constructorul din exemplu atribuie variabilei **x** din clasa **MyClass** valoarea 10. Acest constructor este acum apelat de către **new** la crearea unui nou obiect. De exemplu, în linia:

```
MyClass t1 = new MyClass();
```

se va apela constructorul **MyClass()** pentru obiectul **t1**, atribuind lui **t1.x** valoarea 10. Acelaşi lucru se întâmplă şi pentru **t2**. După instanţiere, **t2.x** are valoarea 10. Rezultatul afişat de program este deci:

```
10 10
```

Constructori cu parametri

În exemplul anterior, s-a utilizat un constructor fără parametri. Chiar dacă aceasta merge în unele situaţii, cel mai adesea aveţi nevoie de un constructor care acceptă unul sau mai mulţi parametri. Puteţi adăuga parametri la un constructor la fel cum îi adăugaţi la orice altă metodă: este de ajuns să-i declaraţi în interiorul parantezelor care urmează după numele constructorului. În exemplul următor, **MyClass()** dispune de un constructor cu un parametru:

```
// Un constructor cu un parametru.

using System;

class MyClass {
  public int x;

  public MyClass(int i) {
    x = i;
  }
}

class ParmConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass(10);
    MyClass t2 = new MyClass(88);

    Console.WriteLine(t1.x + " " + t2.x);
  }
}
```

Rezultatul afişat de program este prezentat mai jos:

```
10 88
```

În această versiune a programului, constructorul **MyClass()** defineşte un parametru numit **i**, care este utilizat la iniţializarea variabilei instanţă **x**. La execuţia liniei:

```
MyClass t1 = new MyClass(10);
```

valoarea 10 va fi transmisă lui **i**, iar apoi atribuită lui **x**.

Adăugarea unui constructor la clasa Vehicle

Putem acum îmbunătăţi clasa **Vehicle** adăugându-i un constructor care iniţializează automat câmpurile **pasageri**, **capcomb** şi **cls** la instanţierea unui obiect. Urmăriţi cu atenţie mai ales modul în care se creează obiectele de tipul **Vehicle**.

```
// Adaugarea unui constructor.

using System;

class Vehicle {
  public int pasageri;   // nr. de pasageri
  public int capcomb;    // capacitatea rezervorului
  public int cls;        // consumul de combustibil, in litri la 100 km

  // Acesta este un constructor pentru clasa Vehicle.
  public Vehicle(int p, int ca, int cs) {
    pasageri = p;
    capcomb = ca;
    cls = cs;
  }

  // Intoarce autonomia.
  public int aut() {
    return (100 * capcomb / cls);
  }

  // Calculeaza combustibilul necesar pe o distanta data.
  public double combnec(int dist) {
    return (double) cls * dist / 100.0;
  }
}

class VehConsDemo {
  public static void Main() {

    // construim vehicule complete
    Vehicle minivan = new Vehicle(7, 80, 15);
    Vehicle sportscar = new Vehicle(2, 70, 20);

    double consum;
    int dist = 407;

    consum = minivan.combnec(dist);

    Console.WriteLine("Pe distanta de " + dist + " km, furgoneta consuma "
                      + consum + " litri de benzina.");

    consum = sportscar.combnec(dist);

    Console.WriteLine("Pe distanta de " + dist + " km, masina sport consuma "
                      + consum + " litri de benzina.");
  }
}
```

Constructor pentru clasa **Vehicle**.

Transmite informaţia despre vehicule clasei **Vehicle**, folosind constructorul ei.

Atât **minivan** cât şi **sportscar** sunt inițializate la creare de către constructorul clasei **Vehicle**. Fiecare obiect este inițializat cu valorile specificate la apelul constructorului său. De exemplu, în linia următoare:

```
Vehicle minivan = new Vehicle(7, 80, 15);
```

valorile 7, 80 şi 15 sunt transmise constructorului clasei **Vehicle()** atunci când **new** creează obiectul. În consecință, cópiile din obiectul **minivan** pentru variabilele **pasageri**, **capcomb** şi **cls** vor conține valorile 7, 80 respectiv 15, rezultatul afişat de program fiind acelaşi ca şi în secțiunea precedentă.

Exerciții la minut

- Ce este un constructor şi când este acesta executat?
- Au constructorii un tip explicit?

Mai multe despre operatorul new

După ce am aflat mai multe despre clase şi despre constructorii acestora, să studiem în amănunt şi operatorul **new**. Operatorul **new** are formatul general:

class-var = **new** *nume-clasa*();

Mai sus, *class-var* reprezintă variabila care va fi creată, având ca tip o clasă. *Nume-clasa* este numele clasei care va fi instanțiată. Numele clasei urmat de paranteze precizează un apel al constructorului clasei. Dacă o clasă nu îşi defineşte propriul constructor, atunci **new** va utiliza constructorul implicit pus la dispoziție de C#. În concluzie, puteți utiliza **new** pentru a crea obiecte având ca tip orice clasă.

Deoarece memoria este o resursă finită, se poate întâmpla ca **new** să nu poată aloca memorie pentru un obiect, atunci când aceasta nu este suficientă. În astfel de situații, se produce o excepție la execuție. (Veți învăța cum să tratați astfel de excepții, precum şi altele, în modulul 9.) Pentru programele prezentate ca exemple în această carte, nu este cazul să vă îngrijorați de lipsa memoriei, dar trebuie să luați în calcul şi această posibilitate în programele reale pe care le veți elabora.

- Un constructor este o metodă care se execută la instanțierea unui obiect dintr-o clasă. Constructorul se utilizează la inițializarea obiectului creat.
- Nu.

Sfatul expertului

Întrebare: De ce nu este cazul să utilizăm new şi pentru variabile de tipuri valorice, cum ar fi int sau float?

Răspuns: În C#, variabilele de tipuri valorice îşi conțin valorile proprii. Memoria necesară stocării acestor valori este alocată în mod automat de către compilator, atunci când programul este compilat. În consecință, nu este nevoie de o alocare explicită a memoriei, utilizând **new**. Invers, variabilele de tip referință memorează referințe către obiecte. Memoria necesară stocării acestor obiecte este alocată dinamic, pe parcursul execuției.

Faptul că tipurile fundamentale, cum sunt **int** sau **char**, nu sunt tipuri referință, îmbunătățeşte cu mult performanțele programelor dumneavoastră. La utilizarea oricărui tip referință, apare un nivel de indirectare care determină o întârziere la fiecare acces la un obiect, întârziere care este evitată în cazul tipurilor valorice.

Ca un lucru interesant, este permisă utilizarea lui **new** cu tipuri valorice, ca în exemplul următor:

```
int i = new int();
```

Aceasta determină invocarea constructorului implicit al tipului **int**, care inițializează pe **i** cu zero. În general, invocarea lui **new** pentru un tip valoric determină apelul constructorului implicit pentru acel tip. Aceasta nu determină însă şi o alocare dinamică a memoriei. De fapt, majoritatea programatorilor nu utilizează **new** pentru tipurile valorice.

Colectarea spațiului neutilizat şi destructori

După cum ați observat, obiectele sunt alocate dinamic dintr-o zonă de memorie liberă utilizând operatorul **new**. Desigur că memoria nu este infinită, iar memoria disponibilă poate fi epuizată. În concluzie, este posibil ca **new** să eşueze, din cauză că nu există suficientă memorie disponibilă pentru a crea obiectul dorit. Din acest motiv, una dintre componentele esențiale ale oricărui mecanism de alocare dinamică este recuperarea memoriei din obiectele care nu mai sunt utilizate, memoria recuperată devenind disponibilă pentru realocarea ulterioară. În multe limbaje de programare, eliberarea memoriei alocate anterior se efectuează manual. De exemplu, în C++, trebuie să utilizați operatorul **delete** pentru a elibera memoria. Limbajul C# utilizează însă o metodă diferită, mai puțin predispusă la apariția erorilor: *colectarea automată* a spațiului disponibil.

Sistemul de colectare automată din C# recuperează automat spațiul ocupat de obiecte – acționând în mod transparent, în spatele scenei, fără a necesita intervenția programatorului. Funcționarea acestui sistem se bazează pe principiul: dacă nu mai

există nici o referință la un obiect, se presupune că acel obiect nu mai este necesar, iar memoria ocupată de el este eliberată. Această memorie reciclată poate fi apoi utilizată pentru o alocare ulterioară.

Colectarea automată se produce destul de rar pe parcursul execuției programului dumneavoastră. Nu se va declanșa doar pentru că există unul sau mai multe obiecte care nu mai sunt folosite. Din rațiuni de eficiență, activitatea de colectare automată se va executa numai atunci când se îndeplinesc următoarele două condiții: există obiecte care trebuie reciclate și există o necesitate strictă de a le recicla. Rețineți că operația de colectare necesită un timp relativ mare; așadar, sistemul C# o va executa numai atunci când este imperios necesar. Nu puteți deci să determinați cu exactitate când are loc colectarea spațiului neutilizat.

Destructori

Există posibilitatea de a defini o metodă care va fi apelată exact înainte de distrugerea finală a obiectului de către sistemul de colectare automată. Această metodă este numită *destructor* și se utilizează pentru a vă asigura că în urma obiectului rămâne „curățenie". Spre exemplu, puteți utiliza un destructor pentru a vă asigura că un fișier deschis deținut de obiectul distrus va fi închis.

Destructorii au forma generală:

```
~nume-clasa() {
    // codul destructorului
}
```

Mai sus, *nume-clasa* reprezintă numele clasei. Destructorul este deci declarat similar cu un constructor, fiind însă precedat de caracterul ~ (tilda). Observați că nici destructorul nu are un tip explicit.

Pentru adăugarea unui destructor la o clasă, îl includeți ca și pe orice alt membru al clasei. Acesta va fi apelat înainte ca un obiect din acea clasă să fie reciclat. În interiorul destructorului, se vor specifica acțiunile care trebuie executate înainte ca obiectul să fie distrus.

Este foarte important să înțelegeți că destructorul se apelează imediat înaintea colectării automate. Acesta nu va fi apelat când obiectul iese din domeniul de valabilitate, de exemplu. (Aceasta diferă de destructorii din C++, care *sunt* apelați când domeniul de valabilitate al obiectului se încheie.) Aceasta înseamnă că nu puteți determina cu exactitate când se va executa un destructor. Mai mult, este posibil ca programul dumneavoastră să se termine înaintea începerii operației de colectare automată, caz în care destructorul nu va fi apelat deloc.

Proiectul 4-2: Prezentarea destructorilor

După cum am explicat, obiectele nu sunt în general reciclate de îndată ce nu mai sunt necesare. Sistemul de colectare automată așteaptă până când își poate desfășura activitatea în mod eficient, de regulă atunci când există mai multe obiecte neutilizate. Pentru a prezenta deci utilizarea unui destructor, aveți nevoie de crearea și distrugerea unui număr mare de obiecte – ceea ce veți și face în cadrul acestui proiect.

Pas cu pas

1. Creați un fișier nou, numit **Destruct.cs**.
2. Creați mai întâi clasa **Destruct** prezentată mai jos:

```
class Destruct {
  int x;

  public Destruct(int i) {
    x = i;
  }

  // apelat la reciclarea obiectului
  ~Destruct() {
    Console.WriteLine("Distrugem " + x);
  }

  // genereaza un obiect care este imediat distrus
  public void generator(int i) {
    Destruct o = new Destruct(i);
  }
}
```

Constructorul atribuie variabilei instanță x o valoare cunoscută. În acest exemplu, **x** este utilizat ca un cod care personalizează obiectul. Destructorul va afișa valoarea lui **x**, atunci când un obiect este reciclat. Extrem de interesantă este metoda **generator()**. Aceasta mai întâi creează un obiect de tipul **Destruct**, pe care îl distruge imediat (deoarece nu mai este referit – n.t.). În pasul următor, veți observa cum funcționează această metodă.

3. Creați clasa **DestructDemo** prezentată mai jos:

```
class DestructDemo {
  public static void Main() {
    int count;

    Destruct ob = new Destruct(0);

    /* Generam acum un numar mare de obiecte. Intr-un anumit moment, se va
       produce colectarea automata.
       Observatie: este posibil sa fie nevoie sa mariti numarul de obiecte
       generate pentru a forta colectarea automata. */

    for(count = 1; count < 100000; count++)
      ob.generator(count);
  }
}
```

Această clasă creează un obiect inițial de tipul **Destruct** numit **ob**. Utilizând obiectul **ob**, se creează încă 100 000 de obiecte, apelând metoda **generator()** a lui **ob**. Aceasta are același efect cu crearea și distrugerea a 100 000 de obiecte. În momente diferite de pe

parcursul acestui proces, va avea loc colectarea spațiului neutilizat. Frecvența exactă a acestei operații depinde de mai mulți factori, printre care cantitatea inițială de memorie disponibilă, sistemul de operare etc. Într-un anumit moment, veți începe să observați mesajele afișate de către destructor. Dacă nu observați nici un mesaj, măriți numărul de obiecte generate crescând contorul din bucla **for**.

4. Iată codul complet al programului **DestructDemo.cs**:

```
/*
    Proiectul 4-2
    Prezinta utilizarea unui destructor.
*/

using System;

class Destruct {
  int x;

  public Destruct(int i) {
    x = i;
  }

  // apelat la reciclarea obiectului
  ~Destruct() {
    Console.WriteLine("Distrugem " + x);
  }

  // genereaza un obiect care este imediat distrus
  public void generator(int i) {
    Destruct o = new Destruct(i);
  }
}

class DestructDemo {
  public static void Main() {
    int count;

    Destruct ob = new Destruct(0);

    /* Generam acum un numar mare de obiecte. Intr-un anumit moment, se va
       produce colectarea automata.
       Observatie: este posibil sa fie nevoie sa mariti numarul de obiecte
       generate pentru a forta colectarea automata. */

    for(count = 1; count < 100000; count++)
      ob.generator(count);
  }
}
```

Cuvântul cheie this

Înainte de a încheia acest modul, este necesar să prezentăm cuvântul cheie **this**. Atunci când o metodă este apelată, aceasta primește un parametru efectiv implicit, care este o referință la obiectul invocat (adică la obiectul asupra căruia a fost apelată metoda). Această referință se numește **this**. Pentru a înțelege ce este **this**, să

considerăm mai întâi un program care creează o clasă **Pwr**, care calculează rezultatul ridicării unui număr la o putere întreagă:

```
using System;

class Pwr {
  public double b;
  public int e;
  public double val;

  public Pwr(double num, int exp) {
    b = num;
    e = exp;

    val = 1;
    if(exp == 0) return;
    for( ; exp > 0; exp--) val = val * b;
  }

  public double get_pwr() {
    return val;
  }
}

class DemoPwr {
  public static void Main() {
    Pwr x = new Pwr(4.0, 2);
    Pwr y = new Pwr(2.5, 1);
    Pwr z = new Pwr(5.7, 0);

    Console.WriteLine(x.b + " ridicat la puterea " + x.e +
                      " este " + x.get_pwr());
    Console.WriteLine(y.b + " ridicat la puterea " + y.e +
                      " este " + y.get_pwr());
    Console.WriteLine(z.b + " ridicat la puterea " + z.e +
                      " este " + z.get_pwr());
  }
}
```

După cum se știe, în interiorul unei metode, ceilalți membri ai clasei pot fi utilizați direct, fără includerea vreun alt calificator corespunzător unui obiect sau unei clase. În interiorul metodei **get_pwr()**, instrucțiunea:

```
return val;
```

precizează că se va întoarce copia lui **val** asociată cu obiectul care a invocat metoda. Aceeași instrucțiune se putea însă rescrie și astfel:

```
return this.val;
```

Aici, **this** se referă la obiectul pentru care a fost apelată metoda **get_pwr()**. În consecință, **this.val** desemnează copia lui **val** din acel obiect. Spre exemplu, când **get_pwr()** este apelată de **x**, **this** din instrucțiunea precedentă corespunde cu **x**. Forma instrucțiunii fără utilizarea lui **this** este aici doar o prescurtare.

Iată codul complet al clasei **Pwr**, rescris utilizând referința **this**:

```
using System;

class Pwr {
  public double b;
  public int e;
  public double val;

  public Pwr(double num, int exp) {
    this.b = num;
    this.e = exp;

    this.val = 1;
    if(exp == 0) return;
    for( ; exp > 0; exp-) this.val = this.val * this.b;
  }

  public double get_pwr() {
    return this.val;
  }
}
```

În realitate, nici un programator în C# nu va scrie clasa **Pwr** ca mai sus, deoarece nu câştigă nimic, forma standard fiind şi mai simplă. Cuvântul **this** are însă alte câteva utilizări importante. De exemplu, sintaxa limbajului C# permite ca numele unui parametru sau al unei variabile locale să coincidă cu numele unei variabile instanţă. În această situaţie, numele local va *ascunde* variabila instanţă. Puteţi avea acces la variabila instanţă care a fost ascunsă referind-o prin intermediul lui **this**. De exemplu, chiar dacă nu recomandăm acest stil, forma de mai jos a constructorului **Pwr()** este corectă din punct de vedere sintactic:

```
public Pwr(double b, int e) {
  this.b = b;
  this.e = e;

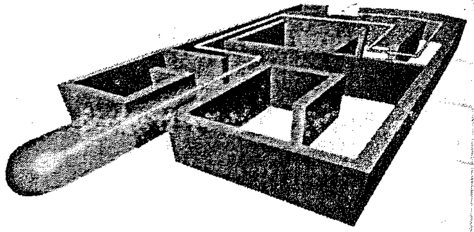
  val = 1;
  if(e == 0) return;
  for( ; e > 0; e--) val = val * b;
}
```

Aici referim variabila instanţă **b**, nu parametrul.

În această versiune, numele parametrilor coincid cu numele variabilelor instanţă, pe care deci le ascund. Cuvântul **this** este însă utilizat pentru a „descoperi" variabilele instanţă.

Verificarea cunoştinţelor

1. Ce diferenţă există între o clasă şi un obiect?
2. Cum poate fi definită o clasă?
3. Ce entităţi au câte o copie în cadrul fiecărui obiect?
4. Utilizând două instrucţiuni separate, arătaţi cum puteţi declara un obiect numit **counter** aparţinând unei clase numite **MyCounter**.
5. Arătaţi cum puteţi declara o metodă numită **myMeth()** care întoarce un rezultat de tipul **double** şi are doi parametri de tipul **int**, numiţi **a** şi **b**.
6. Cum se efectuează revenirea dintr-o metodă care întoarce o valoare?
7. Ce nume trebuie să aibă un constructor?
8. Ce face operatorul **new**?
9. Ce este colectarea spaţiului neutilizat şi cum funcţionează? Ce este un destructor?
10. Ce reprezintă **this**?



Modulul 5

Mai multe despre tipurile de date și operatori

Scopurile acestui modul

- **Însușirea lucrului cu tablouri**
- **Studiul obiectelor de tipul `string`**
- **Utilizarea buclei `foreach`**
- **Studiul operatorilor pe biți**
- **Învățarea operatorului ternar ?**

Acest modul revine asupra subiectului tipurilor de date și operatorilor din C#. Vom discuta despre tablouri, despre tipul **`string`**, operatorii pe biți și operatorul ternar ?. Pe parcurs, vom prezenta și bucla **`foreach`**.

Tablouri

Un tablou reprezintă o colecție de variabile de același tip, referite prin intermediul unui nume comun. În C#, tablourile pot avea una sau mai multe dimensiuni, cel mai frecvent întâlnite fiind cele unidimensionale. Tablourile se utilizează pentru o mare varietate de scopuri, deoarece ele oferă o modalitate convenabilă de a grupa mai multe variabile de același tip la un loc. Spre exemplu, puteți utiliza un tablou pentru a memora maximele termice diurne pe parcursul unei luni, o listă a prețurilor medii pentru anumite acțiuni sau colecția personală de cărți de programare.

Avantajul esențial pe care îl aduc tablourile constă în organizarea datelor într-o manieră în care pot fi gestionate cu ușurință. De exemplu, dacă aveți un tablou care conține veniturile unui grup selectat de familii, este foarte ușor să calculați venitul mediu al grupului efectuând o prelucrare repetitivă asupra tabloului. De asemenea, tablourile organizează datele astfel încât acestea pot fi sortate cu ușurință.

Deși tablourile din C# se pot utiliza la fel ca și tablourile din alte limbaje de programare, ele au o caracteristică aparte: sunt implementate ca obiecte. Acesta este unul dintre motivele pentru care am amânat discuția despre tablouri după prezentarea obiectelor. Prin implementarea tablourilor ca obiecte s-au câștigat câteva avantaje importante, unul dintre ele fiind acela că spațiul ocupat de tablourile neutilizate poate fi colectat automat.

Tablouri unidimensionale

Un tablou unidimensional este o listă de variabile de același tip. Astfel de liste se întâlnesc destul de frecvent în programare. De exemplu, puteți memora într-un tablou unidimensional numerele conturilor utilizatorilor activi dintr-o rețea. Un alt tablou poate fi utilizat la memorarea numărului de goluri marcate de o echipă de fotbal în turul unui campionat.

Pentru a declara un tablou unidimensional, se utilizează forma generală de mai jos:

tip [ ] *nume-tablou* = **new** *tip*[*dimensiune*];

Aici, *tip* reprezintă tipul de bază al tabloului. Tipul de bază determină tipul tuturor elementelor care alcătuiesc tabloul. Remarcați parantezele pătrate care urmează după *tip*. Acestea indică faptul că este vorba de declarația unui tablou unidimensional. Numărul elementelor pe care tabloul le poate memora este precizat de către *dimensiune*. Datorită implementării tablourilor ca obiecte, crearea unui tablou este un proces în doi pași. Mai întâi, se declară o variabilă de referință la tablou. În al doilea rând, se alocă memorie pentru tablou, atribuind apoi variabilei tablou o referință la zona alocată. Tablourile în C# sunt deci alocate dinamic, utilizând operatorul **`new`**.

Observație

Dacă aveți experiență în C sau C++, fiți foarte atent la modul în care se declară tablourile. Mai exact, parantezele pătrate trebuie să urmeze după numele tipului de bază și nu după numele tabloului.

Iată un prim exemplu. Instrucțiunea următoare creează un tablou cu zece elemente de tipul **`int`** pe care îl asociază unei variabile tablou numite **`sample`**:

```
int [] sample = new int[10];
```

Această declarație funcționează la fel ca și declarația oricărui alt obiect. Variabila **`sample`** conține o referință la zona de memorie alocată de **`new`**. Această zonă este suficientă pentru a memora zece elemente de tipul **`int`**.

Ca și în cazul obiectelor, este posibil să separăm declarația anterioară în două părți. De exemplu:

```
int [] sample;
sample = new int[10];
```

În acest caz, atunci când variabila **`sample`** este creată inițial, nu referă nici un obiect fizic. Numai după execuția celei de-a doua instrucțiuni, **`sample`** va fi asociată cu un tablou.

Un element individual din tablou poate fi accesat utilizând un index. Un *index* descrie poziția unui element în cadrul tabloului. În C#, toate tablourile au indexul primului element egal cu zero. Deoarece tabloul **`sample`** are 10 elemente, indecșii în acest tablou vor lua valori între 0 și 9. Pentru a indexa un tablou, precizați numărul

elementului dorit, înconjurat de paranteze pătrate. Primul element din tabloul **sample** este deci **sample[0]**, în timp ce ultimul este **sample[9]**. De exemplu, programul următor va încărca în tabloul **sample** numerele de la 0 la 9:

```
// Prezentarea unui tablou unidimensional

using System;

class ArrayDemo {
  public static void Main() {
    int [] sample = new int[10];
    int i;

    for(i = 0; i < 10; i = i + 1)
      sample[i] = i;

    for(i = 0; i < 10; i = i + 1)
      Console.WriteLine("Elementul sample[" + i + "]: " + sample[i]);
  }
}
```

Tablourile sunt indexate de la 0.

Rezultatul programului este afişat în continuare:

```
Elementul sample[0]: 0
Elementul sample[1]: 1
Elementul sample[2]: 2
Elementul sample[3]: 3
Elementul sample[4]: 4
Elementul sample[5]: 5
Elementul sample[6]: 6
Elementul sample[7]: 7
Elementul sample[8]: 8
Elementul sample[9]: 9
```

Conceptual, tabloul **sample** se prezintă astfel:

0	1	2	3	4	5	6	7	8	9
sample[0]	sample[1]	sample[2]	sample[3]	sample[4]	sample[5]	sample[6]	sample[7]	sample[8]	sample[9]

Tablourile se întrebuinţează des în programare deoarece vă oferă o modalitate uşoară de lucru cu un număr mare de variabile de acelaşi tip. De exemplu, programul din continuare determină valoarea minimă şi cea maximă memorate în tabloul **nums**, parcurgând tabloul cu ajutorul unei bucle **for**:

```
// Determinarea valorii minime si maxime dintr-un tablou.

using System;

class MinMax {
  public static void Main() {
    int [] nums = new int[10];
    int min, max;
```



```
    nums[0] = 99;
    nums[1] = -10;
    nums[2] = 100123;
    nums[3] = 18;
    nums[4] = -978;
    nums[5] = 5623;
    nums[6] = 463;
    nums[7] = -9;
    nums[8] = 287;
    nums[9] = 49;

    min = max = nums[0];
    for(int i = 1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    Console.WriteLine("Minimul si maximul: " + min + " " + max);
  }
}
```

Rezultatul afişat de program este cel prezentat mai jos:

```
Minimul si maximul: -978 100123
```

Iniţializarea unui tablou

În programul anterior, valorile au fost încărcate în tabloul **nums** manual, utilizând zece instrucţiuni de atribuire separate. Deşi este perfect corect, există o modalitate mai simplă de a realiza această operaţie. Tablourile pot fi iniţializate direct de la creare. Forma generală utilizată pentru iniţializarea unui tablou unidimensional este cea prezentată mai jos:

tip [] *nume-tablou* = { *val1*, *val2*, *val3*, ..., *valN* };

Valorile iniţiale au fost specificate prin *val1* până la *valN*. Ele sunt atribuite în secvenţă, de la stânga la dreapta, în ordinea indicilor. Compilatorul de C# alocă în mod automat un tablou suficient de mare pentru a putea memora secvenţa de valori iniţiale specificată. Nu este deci nevoie să utilizaţi operatorul **new**. De exemplu, iată o modalitate mai bună de a scrie programul **MinMax**:

```
// Determinarea valorii minime si maxime dintr-un tablou.

using System;

class MinMax {
  public static void Main() {
    int [] nums = { 99, -10, 100123, 18, -978,
                    5623, 463, -9, 287, 49 };
    int min, max;

    min = max = nums[0];
    for(int i = 1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
```

Secvenţă de iniţializare a unui tablou.

```
    Console.WriteLine("Minimul si maximul: " + min + " " + max);
  }
}
```

Ca fapt divers, chiar dacă nu este nevoie, puteţi totuşi utiliza **new** la iniţializarea unui tablou. Spre exemplu, acesta este un mod corect, dar redundant, de a iniţializa tabloul **nums** din programul precedent.

```
int [] nums = new int[] { 99, -10, 100123, 18, -978,
                          5623, 463, -9, 287, 49 };
```

Chiar dacă aici este redundantă, forma cu **new** a iniţializării unui tablou este utilă atunci când atribuiţi un tablou nou unei variabile referinţă care există deja. De exemplu,

```
int [] nums;
nums = new int[] { 99, -10, 100123, 18, -978,
                   5623, 463, -9, 287, 49 };
```

În acest caz, tabloul **nums** este declarat în prima instrucţiune şi iniţializat în cea de-a doua.

Verificarea marginilor

Marginile tablourilor sunt strict verificate în C#; depăşirea sfârşitului de tablou sau utilizarea indicilor negativi generând erori la execuţie. Dacă doriţi să vă convingeţi singur de aceasta, încercaţi următorul program care depăşeşte în mod intenţionat sfârşitul unui tablou:

```
// Prezinta depasirea unui tablou.

using System;

class ArrayErr {
  public static void Main() {
    int [] sample = new int[10];
    int i;

    // fortam depasirea tabloului
    for(i = 0; i < 100; i = i + 1)
      sample[i] = i;          <-- Depăşeşte sfârşitul tabloului sample.
  }
}
```

De îndată ce **i** atinge valoarea 10, se generează o excepţie de tipul **IndexOutOfRangeException**, iar programul se termină.

Exerciţii la minut

- Tablourile pot fi accesate utilizând un ________ .
- Cum declaraţi un tablou de 10 elemente de tipul **char**?
- În C#, nu se efectuează verificarea depăşirii sfârşitului de tablou la execuţie. Este adevărat sau fals?

Proiectul 5-1: Sortarea unui tablou

Tablourile unidimensionale organizează datele într-o listă liniară indexabilă, constituind astfel o structură de date perfect potrivită pentru sortare. În cadrul acestui proiect, veţi învăţa o modalitate simplă de a sorta un tablou. După cum poate ştiţi, există un număr de algoritmi de sortare diferiţi. Sortarea rapidă, sortarea prin „scuturare" şi metoda lui Shell sunt doar trei dintre aceştia. Cel mai cunoscut şi totodată cel mai simplu şi mai uşor de înţeles dintre algoritmii de sortare este însă *metoda bulelor*. Chiar dacă metoda bulelor nu este foarte eficientă – de fapt, performanţele sale sunt inacceptabile la sortarea tablourilor mari – poate fi utilizată cu rezultate bune la sortarea unor tablouri de dimensiuni mai reduse.

Pas cu pas

1. Creaţi un fişier numit **Bubble.cs**.
2. Metoda bulelor este denumită după modul în care efectuează operaţia de sortare. Metoda efectuează comparaţii repetate şi, dacă este cazul, interschimbă elementele adiacente din tablou. În acest proces, valorile mici se deplasează către un capăt al tabloului, iar cele mari către capătul opus. Procesul este similar din punct de vedere conceptual cu bulele care se aşează pe un anumit nivel într-un rezervor cu apă. Metoda bulelor efectuează mai multe treceri prin tablou, schimbând poziţia elementelor care nu se găsesc la locul lor atunci când este necesar. Numărul de treceri necesar pentru a ne asigura că tabloul este sortat este egal cu numărul elementelor din tablou minus o unitate.
   Prezentăm în continuare codul care reprezintă esenţa metodei bulelor. Tabloul sortat aici se numeşte **nums**.

```
// Sortarea prin metoda bulelor.
for(a = 1; a < size; a++) /
  for(b = size - 1; b >= a; b--) {
    if(nums[b-1] > nums[b]) {  // daca ordinea nu este corecta
      // interschimba elementele adiacente
      t = nums[b-1];
      nums[b-1] = nums[b];
      nums[b] = t;
    }
  }
```

Remarcaţi că sortarea se bazează pe existenţa a două bucle **for**. Bucla interioară verifică elementele adiacente din tablou, căutând elementele care nu respectă ordinea corectă.

- Tablourile pot fi accesate utilizând un index.
- Declaraţia unui tablou de tip **char** cu 10 elemente este următoarea:
  `char [] a = new char[10];`
- Este fals. În C# nu este permisă depăşirea sfârşitului de tablou la execuţie.

La detectarea unei perechi de elemente care nu respectă ordinea, cele două elemente sunt interschimbate. La fiecare pas, elementul minim din cele rămase nesortate se deplasează în locul potrivit. Bucla exterioară determină repetiția acestui proces până la sortarea completă a tabloului.

3. Iată codul complet al programului **Bubble**:

```
/*
   Proiectul 5-1
   Prezinta sortarea prin metoda bulelor.
*/

class Bubble {
  public static void Main() {
    int [] nums = { 99, -10, 100123, 18, -978,
                    5623, 463, -9, 287, 49 };
    int a, b, t;
    int size;

    size = 10; // numarul elementelor de sortat

    // afiseaza tabloul initial
    Console.Write("Tabloul initial este:");
    for(int i = 0; i < size; i++)
      Console.Write(" " + nums[i]);
    Console.WriteLine();

    // Sortarea prin metoda bulelor.
    for(a = 1; a < size; a++)
      for(b = size - 1; b >= a; b--) {
        if(nums[b-1] > nums[b]) { // daca ordinea nu este corecta
          // interschimba elementele adiacente
          t = nums[b-1];
          nums[b-1] = nums[b];
          nums[b] = t;
        }
      }

    // afiseaza tabloul sortat
    Console.Write("Tabloul sortat este:");
    for(int i = 0; i < size; i++)
      Console.Write(" " + nums[i]);
    Console.WriteLine();
  }
}
```

Programul afișează rezultatul prezentat mai jos:

```
Tabloul initial este: 99 -10 100123 18 -978 5623 463 -9 287 49
Tabloul sortat este: -978 -10 -9 18 49 99 287 463 5623 100123
```

4. Deși metoda bulelor este satisfăcătoare pentru tablouri mici, devine ineficientă când dimensiunea tablourilor crește. Cel mai bun algoritm general de sortare este sortarea rapidă. Acesta însă se bazează pe facilități ale limbajului C# pe care încă nu le-ați învățat.

Tablouri multidimensionale

Deși tablourile unidimensionale sunt cel mai frecvent utilizate în programare, nici cele multidimensionale nu constituie, cu siguranță, o prezență izolată. Tablourile multidimensionale au două sau mai multe dimensiuni, iar un element oarecare poate fi accesat utilizând combinația a doi sau mai mulți indici.

Tablouri bidimensionale

Cele mai simple tablouri multidimensionale sunt cele bidimensionale. Într-un tablou bidimensional, poziția unui anumit element este precizată de către doi indici. Un tablou bidimensional poate fi gândit ca o matrice cu informație; unul dintre indici precizează linia, iar celălalt coloana.

Pentru declararea unui tablou bidimensional cu elemente întregi cu dimensiunea de 10 x 20 se poate scrie:

```
int [,] table = new int[10, 20];
```

Fiți foarte atent la această declarație. Remarcați că cele două dimensiuni sunt separate prin virgulă. În prima parte a declarației, sintaxa

```
[,]
```

determină crearea unei variabile de tip referință la un tablou bidimensional. Pentru alocarea efectivă a memoriei pentru tablou utilizând **new**, se folosește sintaxa:

```
int[10, 20]
```

Aceasta creează un tablou cu dimensiunea de 10 x 20, virgula având din nou rolul de a separa cele două dimensiuni.

Pentru accesul la un element dintr-un tablou bidimensional, trebuie să precizați ambii indici, separați prin virgulă. De exemplu, pentru a atribui valoarea 10 elementului din poziția (3, 5) a tabloului **table**, veți scrie:

```
table[3, 5] = 10;
```

Iată un exemplu complet. Acesta încarcă numerele de la 1 la 12 într-un tablou bidimensional, după care afișează conținutul tabloului.

```
// Prezentarea tablourilor bidimensionale.

using System;

class DoiD {
  public static void Main() {
    int t, i;
    int [,] table = new int[3, 4];   // Se declară un tablou bidimensional de 3 x 4.

    for(t = 0; t < 3; ++t) {
      for(i = 0; i < 4; ++i) {
        table[t, i] = (t * 4) + i + 1;
```

```
            Console.Write(table[t, i] + " ");
          }
          Console.WriteLine();
        }
      }
    }
```

În acest exemplu, **table[0, 0]** are valoarea 1, **table[0,1]** are valoarea 2, **table[0, 2]** este 3 ş.a.m.d. Valoarea elementului **table[2, 3]** este 12. Din punct de vedere conceptual, tabloul va arăta similar cu cel din figura 5-1.

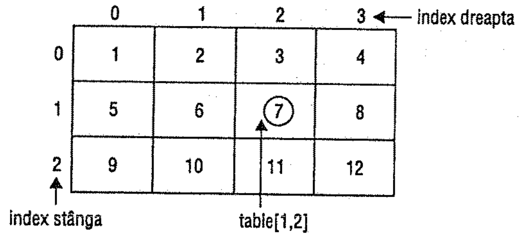


Figura 5-1 O prezentare conceptuală a tabloului **table**, creat de programul **DoiD**.

Observaţie

Dacă aveţi experienţă în C, C++ sau Java, fiţi foarte atent la declararea şi accesul la tablourile multidimensionale. În aceste limbaje, dimensiunile şi indicii tablourilor se specifică în paranteze separate. În C#, dimensiunile sunt separate prin virgule.

Tablouri cu trei sau mai multe dimensiuni

Limbajul C# permite utilizarea tablourilor cu mai mult de două dimensiuni. Forma generală a declaraţiei unui tablou multidimensional este prezentată mai jos:

tip [, ... ,] *nume* = **new** *tip*[*dim1*, *dim2*,..., *dimN*];

De exemplu, declaraţia de mai jos creează un tablou tridimensional de elemente întregi cu dimensiunile 4 x 10 x 3:

```
int [, ,] multidim = new int[4, 10, 3];
```

Pentru a atribui elementului (2, 4, 1) al lui **multidim** valoarea 100, utilizăm instrucţiunea:

```
multidim[2, 4, 1] = 100;
```

Iniţializarea tablourilor multidimensionale

Iniţializarea unui tablou multidimensional se face prin includerea listei de iniţializare a fiecărei dimensiuni într-o pereche proprie de acolade. De exemplu, forma generală a iniţializării unui tablou bidimensional este prezentată în continuare:


```
tip [,] nume-tablou = {
    { val, val, val, ..., val },
    { val, val, val, ..., val },
    ...
    { val, val, val, ..., val }
};
```

Mai sus, *val* indică o valoare iniţială. Fiecare dintre blocurile interioare desemnează o linie din tablou. În cadrul fiecărei linii, prima valoare este memorată în prima poziţie din tablou, a doua valoare în poziţia a doua etc. Remarcaţi că blocurile de iniţializare sunt separate prin virgule, iar acolada finală de închidere este urmată de punct şi virgulă.

De exemplu, programul următor iniţializează un tablou numit **sqrs** cu numerele de la 1 la 10 şi pătratele acestora:

```
// Initializarea unui tablou bidimensional.

using System;

class Patrate {
  public static void Main() {
    int[,] sqrs = {
      { 1, 1 },
      { 2, 4 },
      { 3, 9 },
      { 4, 16 },
      { 5, 25 },
      { 6, 36 },
      { 7, 49 },
      { 8, 64 },
      { 9, 81 },
      { 10, 100 }
    };
    int i, j;

    for(i = 0; i < 10; i++) {
      for(j = 0; j < 2; j++)
        Console.Write(sqrs[i, j] + " ");
      Console.WriteLine();
    }
  }
}
```

Remarcaţi că fiecare rând are propriul set de iniţializatori.

Iată rezultatul afişat de acest program:

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Tablouri „în scară"

În exemplele anterioare, atunci când ați creat un tablou bidimensional, ați creat ceea ce de fapt se numește în C# un *tablou dreptunghiular*. Gândind tablourile bidimensionale ca tabele, tablourile dreptunghiulare corespund cazurilor în care toate liniile din tablou au o lungime constantă. Limbajul C# vă permite însă să creați o categorie specială de tablouri bidimensionale, numite *tablouri „în scară"*. Tablourile în scară sunt *tablouri cu elementele tablouri*, în care lungimea fiecărei linii poate fi diferită. În concluzie, putem utiliza un tablou în scară pentru crearea unei tabele în care lungimile liniilor sunt diferite.

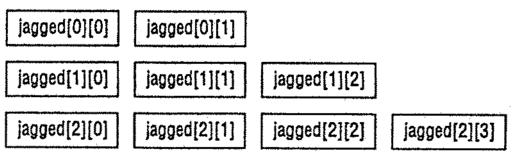
Tablourile în scară sunt declarate utilizând un set separat de paranteze pătrate pentru fiecare dimensiune. De exemplu, pentru declararea unui tablou în scară bidimensional, forma generală utilizată este:

tip [][] *nume-tablou* = **new** *tip*[*dim*][];

Aici, *dim* indică numărul de linii din tablou. Liniile însă nu au fost încă alocate. Acestea se alocă în mod individual, ceea ce permite ca lungimea fiecărei linii să varieze. Spre exemplu, secvența de cod următoare alocă memorie pentru prima dimensiune a tabloului **jagged** atunci când acesta este declarat. Cea de-a doua dimensiune este alocată manual.

```
int [][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[3];
jagged[2] = new int[4];
```

După execuția acestei secvențe, tabloul **jagged** va arăta astfel:



Privind imaginea alăturată, este foarte ușor de intuit de unde vine numele de tablouri în scară.

După ce ați creat un tablou în scară, puteți accesa orice element al său, specificând fiecare index în interiorul unei perechi separate de paranteze pătrate. Spre exemplu, pentru a atribui valoarea 10 elementului (2, 1) din tabloul **jagged**, se utilizează instrucțiunea:

```
jagged[2][1] = 10;
```

Observați că sintaxa este diferită față de cea utilizată pentru accesul la un element dintr-un tablou dreptunghiular.

În continuare, prezentăm un exemplu care utilizează un tablou în scară bidimensional. Să presupunem că scrieți un program care memorează numărul pasagerilor care utilizează trenul unui aeroport. Dacă trenul funcționează de zece ori pe zi în cursul săptămânii și de două ori pe zi sâmbăta și duminica, puteți utiliza tabloul **pasageri** prezentat în programul următor pentru memorarea informației. Remarcați că lungimea celei de-a doua dimensiuni pentru primele cinci linii este 10, iar pentru ultimele două este 2.

```
// Prezinta tablourile in scara.

using System;

class Scara {
  public static void Main() {
    int [][] pasageri = new int [7][];
    pasageri[0] = new int[10];
    pasageri[1] = new int[10];
    pasageri[2] = new int[10];
    pasageri[3] = new int[10];
    pasageri[4] = new int[10];

    pasageri[5] = new int[2];
    pasageri[6] = new int[2];

    int i, j;

    // producem niste date arbitrare
    for(i = 0; i < 5; i++)
      for(j = 0; j < 10; j++)
        pasageri[i][j] = i + j +10;
    for(i = 5; i < 7; i++)
      for(j = 0; j < 2; j++)
        pasageri[i][j] = i + j +10;

    Console.WriteLine("Pasagerii pe fiecare cursa din timpul saptamanii: ");
    for(i = 0; i < 5; i++) {
      for(j = 0; j < 10; j++)
        Console.Write(pasageri[i][j] + " ");
      Console.WriteLine();
    }
    Console.WriteLine();

    Console.WriteLine("Pasagerii pe fiecare cursa din weekend: ");
    for(i = 5; i < 7; i++) {
      for(j = 0; j < 2; j++)
        Console.Write(pasageri[i][j] + " ");
      Console.WriteLine();
    }
  }
}
```

Aici, a doua dimensiune are lungimea de 10...

...dar aici de numai 2.

Majoritatea aplicațiilor nu utilizează tablouri în scară, dar acestea sunt de un real ajutor în anumite situații. De exemplu, dacă aveți nevoie de un tablou bidimensional

de dimensiuni foarte mari, care este însă rar (adică nu toate elementele sale sunt semnficative), tablourile în scară reprezintă soluția ideală.

Exerciții la minut

- Cum se specifică fiecare dimensiune a unui tablou multidimensional dreptunghiular?
- Într-un tablou în scară, pot fi lungimile liniilor diferite între ele?
- Cum se inițializează tablourile multidimensionale?

Atribuirea referințelor la tablouri

Ca și în cazul celorlalte obiecte, atunci când atribuiți o variabilă de tipul referință la un tablou alteia, se modifică pur și simplu obiectul referit de către variabilă. Nu se produce o copiere a tabloului și nici conținutul unui tablou nu se copiază în celălalt. De exemplu, să considerăm programul:

```
// Atribuirea referintelor la tablouri.

using System;

class AtribRef {
  public static void Main() {
    int i;

    int [] nums1 = new int[10];
    int [] nums2 = new int [10];

    for(i = 0; i < 10; i++) nums1[i] = i;

    for(i = 0; i < 10; i++) nums2[i] = -i;

    Console.Write("Tabloul nums1: ");
    for(i = 0; i < 10; i++)
      Console.Write(nums1[i] + " ");
    Console.WriteLine();

    Console.Write("Tabloul nums2: ");
    for(i = 0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

    nums2 = nums1; // acum, nums2 refera pe nums1
```

- Dimensiunile sunt specificate între paranteze pătrate, fiind separate prin virgulă.
- Da.
- Tablourile multidimensionale se inițializează incluzând inițializatorii fiecărui subtablou într-un set propriu de acolade.


```
    Console.Write("Tabloul nums2 dupa atribuire: ");
    for(i = 0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();

    // modificam tabloul nums1 prin intermediul lui nums2
    nums2[3] = 99;

    Console.Write("Tabloul nums1 dupa modificarea prin intermediul lui nums2: ");
    for(i = 0; i < 10; i++)
      Console.Write(nums2[i] + " ");
    Console.WriteLine();
  }
}
```

Programul afișează rezultatul prezentat mai jos:

```
Tabloul nums1: 0 1 2 3 4 5 6 7 8 9
Tabloul nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Tabloul nums2 dupa atribuire: 0 1 2 3 4 5 6 7 8 9
Tabloul nums1 dupa modificarea prin intermediul lui nums2: 0 1 2 99 4 5 6 7 8 9
```

După cum se observă examinând acest rezultat, după atribuirea lui `nums1` către `nums2`, ambele variabile referință sunt asociate cu același obiect.

Utilizarea proprietății Length

O serie de avantaje sunt rezultatul implementării tablourilor în C# ca obiecte. Unul dintre acestea provine din faptul că fiecare tablou are asociată o proprietate numită `Length`, care conține numărul de elemente pe care tabloul le poate memora. Fiecare tablou conține deci un câmp care memorează lungimea tabloului. Iată un program care utilizează această proprietate:

```
// Utilizarea proprietatii Length a unui tablou.

using System;

class LengthDemo {
  public static void Main() {
    int [] list = new int[10];
    int [] nums = { 1, 2, 3 };

    int [][] table = new int [3] [];  // tablou de lungime variabila
    // adaugarea celei de-a doua dimensiuni
    table[0] = new int [] { 1, 2, 3 };
    table[1] = new int [] { 4, 5 };
    table[2] = new int [] { 6, 7, 8, 9 };

    Console.WriteLine("Lungimea tabloului list este " + list.Length);
    Console.WriteLine("Lungimea tabloului nums este " + nums.Length);
    Console.WriteLine("Lungimea tabloului table este " + table.Length);
    Console.WriteLine("Lungimea lui table[0] este " + table[0].Length);
    Console.WriteLine("Lungimea lui table[1] este " + table[1].Length);
    Console.WriteLine("Lungimea lui table[2] este " + table[2].Length);
    Console.WriteLine();
```

```
    // utilizarea lungimii la initializarea lui list
    for(int i = 0; i < list.Length; i++)
      list[i] = i * i;

    Console.Write("Tabloul list: ");
    // utilizam lungimea la afisarea lui list
    for(int i = 0; i < list.Length; i++)
      Console.Write(list[i] + " ");
    Console.WriteLine();
  }
}
```

Acest program va afişa următorul rezultat:

```
Lungimea tabloului list este 10
Lungimea tabloului nums este 3
Lungimea tabloului table este 3
Lungimea lui table[0] este 3
Lungimea lui table[1] este 2
Lungimea lui table[2] este 4

Tabloul list: 0 1 4 9 16 25 36 49 64 81
```

Reţineţi în special modul de utilizare a proprietăţii **Length** pentru tabloul bidimensional în scară **table**. După cum s-a precizat, un tablou în scară bidimensional este un tablou de tablouri. Prin urmare, expresia:

```
table.Length
```

întoarce numărul de *tablouri* memorate în **table**, adică 3 în acest caz. Pentru a obţine lungimea fiecărui subtablou individual al lui **table**, se vor utiliza expresii de forma:

```
table[0].Length
```

care, în acest caz, determină lungimea primului subtablou.

Un alt amănunt care merită reţinut din programul **LengthDemo** este modul în care valoarea **list.Length** este utilizată în buclele **for** pentru a stabili numărul iteraţiilor care se vor efectua. Deoarece fiecare tablou îşi memorează propria lungime, este mai indicat să utilizaţi această informaţie decât să reţineţi pe dinafară dimensiunea tabloului. Reţineţi însă că valoarea lui **Length** nu are nimic în comun cu numărul de elemente efectiv utilizate. **Length** conţine numărul maxim de elemente pe care tabloul le poate memora.

Prezenţa proprietăţii **Length** simplifică mulţi algoritmi, deoarece face ca anumite operaţii asupra tablourilor să fie mai uşor şi mai sigur de executat. De exemplu, următorul program utilizează **Length** pentru a copia un tablou în altul, evitând în acelaşi timp depăşirea şi excepţia la execuţie pe care aceasta o generează:

```
// Utilizarea proprietatii Length la copierea unui tablou.

using System;

class Copiere {
  public static void Main() {
```



```
    int i;
    int [] nums1 = new int[10];
    int [] nums2 = new int[10];

    for(i = 0; i < nums1.Length; i++) nums1[i] = i;

    // copiem nums1 in nums2
    if(nums2.Length >= nums1.Length)
      for(i = 0; i < nums1.Length; i++)
        nums2[i] = nums1[i];

    for(i = 0; i < nums2.Length; i++)
      Console.WriteLine(nums2[i] + " ");
  }
}
```

Aici, **Length** ne ajută să efectuăm două operaţii importante. Mai întâi, se utilizează pentru a confirma că tabloul destinaţie este suficient de mare pentru a memora conţinutul tabloului sursă. În al doilea rând, se foloseşte la stabilirea condiţiei de terminare a buclei **for** care efectuează copierea. Desigur că, în acest exemplu simplu, dimensiunile tablourilor sunt uşor de determinat, dar aceeaşi metodă se poate aplica într-o multitudine de situaţii mai dificile.

Exerciţii la minut

- La atribuirea unei referinţe către un tablou alteia, elementele din primul tablou se copiază într-al doilea. Este adevărat sau fals?
- Relativ la tablouri, ce reprezintă **Length**?

Proiectul 5-2: Implementarea unei cozi

După cum poate ştiţi, o *structură de date* reprezintă un mijloc de organizare a datelor. Cea mai simplă structură de date este tabloul, o listă liniară care permite accesul aleator la elementele sale. Tablourile sunt deseori utilizate ca suport pentru implementarea unor structuri mai sofisticate, cum sunt stivele şi cozile. *Stivele* sunt liste în care elementele pot fi accesate numai în ordinea „primul sosit, ultimul servit" (FILO). *Cozile* sunt liste care permit accesul la elemente numai în ordinea „primul sosit, primul servit" (FIFO). Aşadar, o stivă se aseamănă cu un teanc de farfurii pe o masă, farfuria aflată pe masă fiind ultima care va fi utilizată. O coadă este similară unui rând de aşteptare la o bancă; primul client din rând este primul servit.

Motivul care determină un interes crescut pentru structuri de date cum sunt stivele şi cozile este faptul că ele combină memorarea informaţiei cu metode de acces la informaţia memorată. Stivele şi cozile sunt deci *motoare de date* care asigură ele însele memorarea şi regăsirea informaţiei, sarcini de care programul dumneavoastră este degrevat. În mod

---

- Este fals. Se modifică numai referinţa.
- **Length** este o proprietate comună tuturor tablourilor. Ea conţine numărul elementelor care pot fi memorate în tablou.

evident, o astfel de combinație se implementează excelent cu ajutorul unei clase. În cadrul acestui proiect, veți crea o clasă care implementează o coadă.

În general, cozile permit două operații de bază: *put* și *get*. Fiecare operație *put* adaugă un nou element la sfârșitul cozii. Orice operație *get* regăsește următorul element de la începutul cozii. Operațiile asupra cozilor consumă elementele asupra cărora acționează. Dacă un element a fost regăsit, nu mai poate fi regăsit încă o dată. O coadă poate fie să se umple, dacă nu mai rămâne spațiu disponibil pentru memorarea unui nou element, fie să devină vidă, dacă toate elementele au fost șterse.

Un ultim aspect: există două tipuri principale de cozi – circulare și liniare. O coadă *circulară* reutilizează pozițiile din tabloul în care este implementată după ștergerea elementelor din coadă. Coada *liniară* în schimb nu le reutilizează și din această cauză poate ajunge la epuizarea spațiului de care dispune. Din motive de simpltate, exemplul din continuare creează o coadă liniară, pe care însă cu un efort minim o puteți transforma într-o coadă circulară.

Pas cu pas

1. Creați un fișier nou, numit **QDemo.cs**.
2. Deși există și alte modalități de a implementa o coadă, metoda pe care o utilizăm aici se bazează pe un tablou.Vom utiliza așadar un tablou pentru a memora elementele care sunt inserate în coadă. Acest tablou va fi accesat utilizând doi indici. Indicele *put* determină poziția în care va fi inserat următorul element. Indicele *get* precizează poziția următorului element care va fi extras din coadă. Rețineți că operația *get* „consumă" elementul pe care îl extrage, deci nu este posibil să extrageți același element de două ori. Coada pe care o vom crea aici memorează caractere, dar același raționament poate fi utilizat pentru a memora orice alt tip de obiecte. Începeți crearea clasei **Queue** cu următoarele trei linii:

```
class Queue {
  public char [] q;   // tabloul care implementeaza coada
  public int putloc, getloc;   // indicii put si get
```

3. Constructorul clasei **Queue** va crea o coadă de dimensiune precizată. Iată codul constructorului **Queue**:

```
public Queue(int size) {
  q = new char[size + 1];   // se aloca memorie pentru coada
  putloc = getloc = 0;
}
```

Remarcați că dimensiunea efectivă a cozii create depășește cu o unitate valoarea specificată prin **size**. Din cauza modului de implementare a cozii, o locație din tablou va rămâne neutilizată; așadar, tabloul creat trebuie să fie cu o unitate mai mare decât dimensiunea precizată a cozii. Indicii *put* și *get* sunt inițializați la zero.

4. Metoda **put()**, care inserează elemente în coadă, este prezentată în continuare:

```
// insereaza un caracter in coada
public void put(char ch) {
  if(putloc == q.Length - 1) {
    Console.WriteLine(" - Coada este plina.");
    return;
  }

  putloc++;
  q[putloc] = ch;
}
```


Metoda verifică la început condiția de „coadă plină". Atunci când **putloc** ajunge egal cu indicele ultimei poziții dn tabloul **q**, înseamnă că nu mai este spațiu pentru a insera noi elemente. În caz contrar, **putloc** este incrementat, iar elementul este memorat în noua locație indicată de **putloc**. **Putloc** reprezintă deci întotdeauna indicele ultimului element inserat.

5. Pentru regăsirea elementelor, utilizați metoda **get()**, prezentată în continuare:

```
// intoarce un caracter din coada
public char get() {
  if(getloc == putloc) {
    Console.WriteLine(" - Coada este vida.");
    return (char) 0;
  }

  getloc++;
  return q[getloc];
}
```

Remarcați mai întâi verificarea condiției de „coadă vidă". Dacă atât **getloc**, cât și **putloc** indică ambele același element, atunci coada este considerată vidă. Din acest motiv, atât **getloc**, cât și **putloc** au fost inițializate cu zero de către constructorul **Queue**. În continuare, **getloc** este incrementat și următorul element este întors. În concluzie, **getloc** arată întotdeauna locația ultimului element extras din coadă.

6. Iată și programul **QDemo.cs** complet:

```
/*
   Proiectul 5-2

   O clasa care reprezinta o coada de caractere.
*/

using System;

class Queue {
  public char [] q;   // tabloul care implementeaza coada
  public int putloc, getloc;   // indicii put si get

  public Queue(int size) {
    q = new char[size + 1];   // se aloca memorie pentru coada
    putloc = getloc = 0;
  }

  // insereaza un caracter in coada
  public void put(char ch) {
    if(putloc == q.Length - 1) {
      Console.WriteLine(" - Coada este plina.");
      return;
    }

    putloc++;
    q[putloc] = ch;
  }

  // intoarce un caracter din coada
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine(" - Coada este vida.");
```

```
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}

// Prezinta utilizarea clasei Queue.
class QDemo {
  public static void Main() {
    Queue bigQ = new Queue(100);
    Queue smallQ = new Queue(4);
    char ch;
    int i;

    Console.WriteLine("Utilizam bigQ pentru memorarea alfabetului.");
    // inseram valori in bigQ
    for(i = 0; i < 26; i++)
      bigQ.put((char) ('A' + i));

    // extragem si afisam elementele din bigQ
    Console.Write("Continutul cozii bigQ: ");
    for(i = 0; i < 26; i++) {
      ch = bigQ.get();
      if(ch != (char)0) Console.Write(ch);
    }
    Console.WriteLine("\n");

    Console.WriteLine("Utilizam smallQ pentru a genera erori.");
    // Utilizam acum smallQ pentru a genera cateva erori
    for(i = 0; i < 5; i++) {
      Console.Write("Incercam sa memoram " + (char)('Z' - i));
      smallQ.put((char) ('Z' - i));
      Console.WriteLine();
    }

    // Generam mai multe erori
    Console.Write("Continutul cozii smallQ: ");
    for(i = 0; i < 5; i++) {
      ch = smallQ.get();

      if(ch != (char)0) Console.Write(ch);
    }
  }
}
```

7. Programul va afișa rezultatul prezentat mai jos:

```
Utilizam bigQ pentru memorarea alfabetului.
Continutul cozii bigQ: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Utilizam smallQ pentru a genera erori.
Incercam sa memoram Z
Incercam sa memoram Y
Incercam sa memoram X
Incercam sa memoram W
```



```
Incercam sa memoram V -- Coada este plina.

Continutul cozii smallQ: ZYXW -- Coada este vida.
```

8. Încercați acum în mod independent să modificați clasa **Queue** astfel încât să memoreze și alte tipuri de obiecte. De exemplu, memorați date de tipul **int** sau **double**.

Bucla foreach

În modulul 3, s-a precizat că limbajul C# definește o buclă denumită **foreach**, pe care însă am amânat să o prezentăm până acum.

Bucla **foreach** se utilizează pentru ciclarea prin elementele unei *colecții*. O colecție reprezintă un grup de obiecte. Limbajul C# definește mai multe tipuri de colecții, unul din aceste tipuri fiind tablourile. Forma generală a lui **foreach** este prezentată mai jos:

foreach(*tip nume-var* **in** *colectie*) *instructiune*;

Aici, *tip nume-var* specifică tipul și numele unei variabile de iterare, care va primi la fiecare iterație a lui **foreach** valoarea unui element din colecție. Colecția prin care se efectuează ciclarea este specificată de *colectie*, în acest caz fiind un tablou. Ca urmare, *tip* trebuie să coincidă (sau să fie compatibil cu) tipul de bază al tabloului. Un amănunt important de reținut este că variabila de iterare nu poate fi modificată cât timp este utilizată pentru accesul la tablou. În consecință, nu puteți actualiza conținutul tabloului atribuind variabilei de iterare o nouă valoare.

În continuare, prezentăm un exemplu simplu care utilizează bucla **foreach**. Acesta creează un tablou de întregi pe care îl și inițializează. Programul afișează apoi valorile din tablou, calculând în același timp și suma lor.

```
// Utilizarea lui foreach.
using System;

class ForeachDemo {
  public static void Main() {
    int sum = 0;
    int [] nums = new int[10];

    // se introduc valori in tabloul nums
    for(int i = 0; i < 10; i++)
      nums[i] = i;

    // utilizam foreach pentru afisarea si insumarea valorilor
    foreach (int x in nums) {    <-- Ciclează prin tabloul nums utilizând o buclă foreach.
      Console.WriteLine("Valoarea este: " + x);
      sum += x;
    }
    Console.WriteLine("Suma este: " + sum);
  }
}
```

Programul afişează rezultatul prezentat mai jos:

```
Valoarea este: 0
Valoarea este: 1
Valoarea este: 2
Valoarea este: 3
Valoarea este: 4
Valoarea este: 5
Valoarea este: 6
Valoarea este: 7
Valoarea este: 8
Valoarea este: 9
Suma este: 45
```

După cum arată şi rezultatul, instrucţiunea **foreach** parcurge elementele tabloului în secvenţă, începând de la indicele minim şi până la cel maxim.

Bucla **foreach** funcţionează şi pentru tablouri multidimensionale. Elementele acestora sunt parcurse în ordinea liniilor, de la primul până la ultimul.

```
// Utilizarea lui foreach pe un tablou bidimensional.
using System;

class ForeachDemo2 {
  public static void Main() {
    int sum = 0;
    int [,] nums = new int[3, 5];

    // se introduc valori in matricea nums
    for(int i = 0; i < 3; i++)
      for(int j = 0; j < 5; j++)
        nums[i, j] = (i+1) * (j+1);

    // utilizarea lui foreach la afisarea si insumarea valorilor
    foreach (int x in nums) {
      Console.WriteLine("Valoarea este: " + x);
      sum += x;
    }
    Console.WriteLine("Suma este: " + sum);
  }
}
```

Rezultatul afişat de program este prezentat mai jos:

```
Valoarea este: 1
Valoarea este: 2
Valoarea este: 3
Valoarea este: 4
Valoarea este: 5
Valoarea este: 2
Valoarea este: 4
Valoarea este: 6
Valoarea este: 8
Valoarea este: 10
Valoarea este: 3
Valoarea este: 6
Valoarea este: 9
Valoarea este: 12
```



```
Valoarea este: 15
Suma este: 90
```

Deoarece **foreach** parcurge elementele unui tablou numai de la începutul acestuia până la sfârşit, puteţi crede în mod greşit că utilitatea sa este limitată. Există un număr mare de algoritmi care necesită exact acest mecanism. De exemplu, în continuare prezentăm o altă modalitate de a scrie clasa **MinMax** pe care am utilizat-o anterior în acest modul la determinarea minimului şi maximului dintr-un set de valori:

```
/* Determinarea valorilor minima si maxima dintr-un tablou
utilizand bucla foreach. */

using System;

class MinMax {
  public static void Main() {
    int [] nums = { 99, -10, 100123, 18, -978,
                    5623, 463, -9, 287, 49 };

    int min, max;

    min = max = nums[0];
    foreach(int val in nums) {
      if(val < min) min = val;
      if(val > max) max = val;
    }
    Console.WriteLine("Minimul si maximul: " + min + " " + max);
  }
}
```

Bucla **foreach** reprezintă o alegere ideală pentru această aplicaţie, deoarece determinarea valorilor minimă şi maximă necesită examinarea fiecărui element. Printre alte genuri de aplicaţii care pot utiliza **foreach** se găsesc calculul unei valori medii, căutarea într-o listă sau copierea unui tablou.

Exerciţii la minut

- Ce face instrucţiunea **foreach**?
- Este posibil ca **foreach** să acceseze elementele unui tablou în ordine inversă?
- Puteţi utiliza **foreach** pentru atribuirea unei valori unui element dintr-un tablou?

- Bucla **foreach** parcurge în ordine toate elementele unui tablou.
- Nu.
- Nu.

Stringuri

Din punctul de vedere al utilităţii de zi cu zi, unul dintre tipurile de date cele mai importante în C# este tipul **string**. Tipul **string** defineşte şi implementează şirurile de caractere. În multe alte limbaje de programare, stringurile sunt reprezentate ca simple tablouri de caractere. Aceasta nu se întâmplă însă şi în C#. În C#, stringurile sunt obiecte. Tipul **string** este deci un tip referinţă.

De fapt, aţi beneficiat de clasa **string** încă din modulul 1, fără însă să vă daţi seama. Orice literal **string** pe care îl creaţi este de fapt un obiect de tipul **string**. Spre exemplu, în instrucţiunea:

```
Console.WriteLine("In C#, stringurile sunt obiecte.");
```

şirul „In C#, stringurile sunt obiecte" este creat implicit ca un obiect de tip **string** de către C#. Utilizarea clasei **string** s-a realizat deci „pe ascuns" în programele de până acum. În această secţiune, veţi învăţa cum puteţi lucra în mod explicit cu stringuri. Vă aducem însă la cunoştinţă că, din cauza dimensiunii relativ mari a clasei **string**, prezentarea noastră este doar introductivă. Această clasă este mult mai adecvată pentru studiul individual.

Construcţia stringurilor

Cel mai simplu mod de a construi un **string** este prin utilizarea unui literal. De exemplu, mai jos **str** reprezintă o variabilă referinţă de tipul **string**, căreia i se atribuie o referinţă la un literal **string**:

```
string str = "Stringurile din C# ofera facilitati puternice.";
```

În acest caz, variabila **str** este iniţializată cu secvenţa de caractere „Stringurile din C# ofera facilitati puternice."

Puteţi de asemenea să creaţi un string pornind de la un tablou de tipul **char**. Spre exemplu:

```
char [] chtab = { 't', 'e', 's', 't' };
string str = new string(chtab);
```

După ce aţi creat un obiect de tipul **string**, îl puteţi utiliza oriunde în locul unui şir de caractere între ghilimele. Spre exemplu, un obiect de tipul **string** poate fi transmis ca parametru funcţiei **WriteLine()**, după cum se observă şi în exemplul următor:

```
// Prezentarea stringurilor.
using System;

class StringDemo {
  public static void Main() {

    char [] chtab = {'U', 'n', ' ', 's', 'i', 'r', '.' };
    string str1 = new string(chtab);
    string str2 = "Alt sir.";
```

Se construiesc obiecte **string** dintr-un tablou de caractere şi dintr-un literal **string**.


```
    Console.WriteLine(str1);
    Console.WriteLine(str2);
  }
}
```

Rezultatul afişat de program este prezentat mai jos:

```
Un sir.
Alt sir.
```

Lucrul cu stringuri

Clasa **string** conţine mai multe metode care operează asupra stringurilor. Să enumerăm numai câteva dintre acestea:

static string Copy(string *str*)	Întoarce o copie a lui *str*.
int CompareTo(string *str*)	Întoarce o valoare negativă dacă stringul care invocă metoda este mai mic decât *str*, pozitivă dacă este mai mare şi nulă când stringurile sunt egale.
int indexOf(string *str*)	Caută în stringul apelant subşirul specificat prin *str*. Întoarce indicele primei apariţii, sau –1 în caz de eşec.
int LastIndexOf(string *str*)	Caută în stringul apelant subşirul specificat prin *str*. Întoarce indicele ultimei apariţii, sau –1 în caz de eşec.

Tipul **string** cuprinde de asemenea şi proprietatea **Length**, care conţine lungimea şirului de caractere.

Pentru a determina valoarea unui caracter dintr-un **string**, este suficient să utilizaţi un index. Spre exemplu:

```
string str = "test";
Console.WriteLine(str[0]);
```

Instrucţiunea va afişa litera **t**. Ca şi în cazul tablourilor, indicii încep de la zero. Un amănunt important este că nu puteţi atribui o valoare nouă unui caracter dintr-un **string** utilizând un index.

Pentru a verifica egalitatea a două stringuri, se poate utiliza operatorul **==**. În mod obişnuit, atunci când operatorul **==** este aplicat asupra referinţelor la obiecte, se stabileşte dacă ambele referinţe sunt asociate cu acelaşi obiect. Aceasta nu se întâmplă însă şi pentru obiectele de tipul **string**. Atunci când aplicăm **==** pentru două referinţe **string**, se verifică de fapt egalitatea şirurilor de caractere asociate. Acelaşi lucru rămâne valabil şi pentru operatorul **!=** în cazul obiectelor **string**, comparându-se conţinutul acestora. Ceilalţi operatori relaţionali, cum sunt < sau >=, compară însă referinţele, la fel ca şi pentru celelalte tipuri de obiecte.

În continuare, iată un program care prezintă câteva operaţii cu stringuri:

```
// Cateva operatii cu stringuri.

using System;

class StrOps {
```

```
public static void Main() {
  string str1 = "In programarea .NET, C# este liderul.";
  string str2 = string.Copy(str1);
  string str3 = "Stringurile din C# ofera facilitati puternice.";
  int result, idx;

  Console.WriteLine("Lungimea lui str1: " + str1.Length);

  // afiseaza str1, caracter cu caracter.
  for(int i = 0; i < str1.Length; i++)
    Console.Write(str1[i]);
  Console.WriteLine();

  if(str1 == str2)
    Console.WriteLine("str1 == str2");
  else
    Console.WriteLine("str1 != str2");

  if(str1 == str3)
    Console.WriteLine("str1 == str3");
  else
    Console.WriteLine("str1 != str3");

  result = str1.CompareTo(str3);
  if(result == 0)
    Console.WriteLine("str1 si str3 sunt egale");
  else if(result < 0)
    Console.WriteLine("str1 este mai mic decat str3");
  else
    Console.WriteLine("str1 este mai mare decat str3");

  // atribuim un nou string lui str2
  str2 = "Unu Doi Trei Unu";

  idx = str2.IndexOf("Unu");
  Console.WriteLine("Indexul primei aparitii a lui Unu: " + idx);
  idx = str2.LastIndexOf("Unu");
  Console.WriteLine("Indexul ultimei aparitii a lui Unu: " + idx);
 }
}
```

Programul va afişa următorul text:

```
Lungimea lui str1: 37
In programarea .NET, C# este liderul.
str1 == str2
str1 != str3
str1 este mai mic decat str3
Indexul primei aparitii a lui Unu: 0
Indexul ultimei aparitii a lui Unu: 13
```

Două stringuri pot fi *concatenate* (unite) utilizând operatorul +. De exemplu, instrucţiunea:

```
string str1 = "Unu";
string str2 = "Doi";
string str3 = "Trei";
string str4 = str1 + str2 + str3;
```

va iniţializa pe **str4** cu şirul de caractere „UnuDoiTrei“.

Tablouri de stringuri

Ca şi orice alt tip de date, stringurile pot fi grupate laolaltă în tablouri. Fie exemplul:

```
// Prezentarea tablourilor de stringuri.
using System;

class StringTab {
  public static void Main() {
    string [] str = { "Acesta", "este", "un", "test." };

    Console.WriteLine("Tabloul initial: ");
    for(int i = 0; i < str.Length; i++)
      Console.Write(str[i] + " ");
    Console.WriteLine("\n");

    // modificam unele dintre stringuri
    str[1] = "a fost";
    str[3] = "test, de asemenea!";

    Console.WriteLine("Tabloul modificat: ");
    for(int i = 0; i < str.Length; i++)
      Console.Write(str[i] + " ");
  }
}
```

Un tablou de stringuri.

Programul afişează următorul text:

```
Tabloul initial:
Acesta este un test.

Tabloul modificat:
Acesta a fost un test, de asemenea!
```

Stringurile nu se pot modifica

Iată un amănunt care vă poate surprinde: conţinutul unui obiect de tipul **string** nu poate fi modificat. Aceasta înseamnă că, după creare, secvenţa de caractere care alcătuieşte stringul nu poate fi alterată. Această restricţie permite limbajului C# să implementeze mai eficient stringurile. Chiar dacă aceasta pare să fie o piedică serioasă, nu este. Atunci când aveţi nevoie de un **string** care este o variaţie a unuia deja existent, puteţi crea unul nou care să conţină modificările dorite. Cum obiectele **string** neutilizate sunt colectate automat, soarta lor nu trebuie să vă preocupe câtuşi de puţin.

Trebuie să stabilim însă cât se poate de clar că variabilele referinţă de tipul **string** îşi pot, desigur, schimba obiectul referit. Doar conţinutul unui anumit obiect de tipul **string** nu mai poate fi modificat după ce a fost creat.

Pentru a înţelege pe deplin de ce imuabilitatea stringurilor nu reprezintă o piedică, vom utiliza o altă metodă a clasei **string**, şi anume **Substring()**. Metoda **Substring()** întoarce un nou şir de caractere care conţine o porţiune precizată din stringul apelant. Datorită faptului că metoda construieşte un nou obiect de tipul **string** în care memorează subşirul, stringul iniţial rămâne nealterat, respectându-se astfel

regula imuabilității. Forma metodei **Substring()** pe care o vom utiliza este prezentată mai jos:

string Substring (int *startIndex*, **int** *len*)

Mai sus, *startIndex* precizează indicele de început, iar *len* lungimea subșirului extras.

Iată un program care prezintă metoda **Substring()** și principiul imuabilității stringurilor:

```
// Utilizarea Substring().
  using System;

class SubStr {
  public static void Main() {
    string orgstr = "Lucrul cu stringuri in C# este simplu.");

    // se construieste un subsir
    string substr = orgstr.Substring(5, 14);

    Console.WriteLine("orgstr: " + orgstr);
    Console.WriteLine("substr: " + substr);
  }
}
```

Se creează un nou **string** care conține subșirul dorit.

Programul afișează textul de mai jos:

```
orgstr: Lucrul cu stringuri in C# este simplu.
substr: l cu stringuri
```

După cum se observă, șirul inițial **orgstr** rămâne nemodificat, în timp ce **substr** va conține subșirul selectat.

Exerciții la minut

- În C#, toate stringurile sunt obiecte. Este adevărat sau fals?
- Cum puteți determina lungimea unui **string**?
- Ce face metoda **Substring()** ?

Sfatul expertului

Întrebare: Ați precizat că, odată create, obiectele de tipul string nu pot fi modificate. Am înțeles că, din punct de vedere practic, aceasta nu constituie o restricție serioasă. Cum se poate însă proceda dacă dorim să creăm stringuri care *pot* fi modificate?

- Este adevărat.
- Lungimea unui **string** se poate obține utilizând proprietatea **Length**.
- Metoda **Substring()** construiește un nou string care reprezintă un subșir al celui pentru care metoda este invocată. Metoda întoarce noul șir construit.

Răspuns: Aveți noroc. Limbajul C# pune la dispoziție o clasă numită **StringBuilder**, care aparține spațiului de nume **System.Text**. Aceasta permite crearea unor obiecte care pot fi modificate. În majoritatea cazurilor însă, veți utiliza **string** și nu **StringBuilder**.

Operatori la nivel de biți

În modulul 2, ați învățat care sunt operatorii aritmetici, relaționali și logici în C#. Chiar dacă aceștia sunt cel mai frecvent utilizați, limbajul C# oferă suplimentar și alți operatori care lărgesc aria problemelor pe care le poate rezolva: operatorii la nivel de biți. Operatorii la nivel de biți acționează direct asupra biților operanzilor lor. Aceștia sunt definiți numai pentru operanzi de tip întreg. Ei nu pot fi utilizați asupra tipurilor **bool, float, double**, sau asupra tipurilor clasă.

Acești operatori sunt denumiți operatori *la nivel de biți* deoarece se utilizează la testarea, modificarea sau deplasarea biților care alcătuiesc o valoare întreagă. Operațiile pe biți sunt importante într-o gamă largă de activități de programare la nivel scăzut, ca de exemplu interogarea sau construirea informației de stare pentru un dispozitiv fizic. Tabelul 5-1 prezintă o listă a operatorilor la nivel de biți.

Operatorii ȘI, SAU, XOR și NON pe biți

Operatorii pe biți ȘI, SAU, XOR și NON sunt &, |, ^, respectiv ~. Aceștia efectuează aceleași operații ca și operatorii logici echivalenți cu ei, prezentați în modulul 2. Diferența este că operatorii pe biți funcționează bit cu bit. Tabela de mai jos prezintă rezultatele tuturor operațiilor, utilizând cifrele binare 1 și 0:

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Raportându-ne la utilizarea cea mai frecventă, putem gândi operația ȘI la nivel de biți ca o modalitate de a face anumiți biți egali cu zero. Aceasta deoarece orice bit 0 în oricare din operanzi va determina ca bitul corespunzător din rezultat să fie tot 0. De exemplu, fie operația:

```
  1 1 0 1 0 0 1 1
  1 0 1 0 1 0 1 0
& ----------------
  1 0 0 0 0 0 1 0
```

Tabelul 5-1 Operatorii la nivel de biţi

Operatorul	Semnificaţia
&	ŞI pe biţi
|	SAU pe biţi
^	SAU exclusiv (XOR) pe biţi
>>	Deplasare spre dreapta
<<	Deplasare spre stânga
~	Complement faţă de 1 (NON)

Programul din continuare utilizează operatorul & la transformarea literelor mici în majuscule prin modificarea la zero a celui de-al şaselea bit. Din definiţia setului de caractere ASCII/Unicode, rezultă că literele mici au valori cu 32 superioare celor mari. În consecinţă, pentru a transforma o literă mică în majusculă, este suficient să facem al şaselea bit egal cu zero, după cum arată şi acest program:

```
// Transformarea in litere mari.
using System;

class UpCase {
  public static void Main() {
    char ch;

    for(int i = 0; i < 10; i++) {
      ch = (char) ('a' + i);
      Console.Write(ch);

      // facem bitul 6 egal cu zero.
      ch = (char) (ch & 65503); // ch este acum majuscula

      Console.Write(ch + " ");
    }
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
aA bB cC dD eE fF gG hH iI jJ
```

Valoarea 65 503 utilizată în instrucţiunea ŞI se reprezintă în binar ca 1111 1111 1101 1111. În consecinţă, operaţia ŞI va lăsa toţi biţii lui **ch** nemodificaţi, cu excepţia celui de-al şaselea, care este anulat.

Operatorul ŞI este foarte util şi atunci când doriţi să determinaţi dacă un anumit bit este 1 sau 0. De exemplu, această instrucţiune determină dacă bitul 4 al lui **status** este 1:

```
if((status & 8) != 0) Console.WriteLine("bitul 4 este 1");
```

Motivul pentru care se utilizează 8 este că în binar, acesta corespunde unei valori care are numai bitul 4 egal cu 1. În concluzie, instrucţiunea **if** se execută numai

dacă şi bitul 4 al lui **status** este 1. O aplicaţie interesantă a acestei idei este afişarea unei valori de tipul **byte** în format binar:

```
// Afisarea bitilor dintr-un byte.
using System;

class ShowBits {
  public static void Main() {
    int t;
    byte val;

    val = 123;
    for(t = 128; t > 0; t = t/2) {
      if((val & t) != 0) Console.Write("1 ");
      else Console.Write("0 ");
    }
  }
}
```

Afişarea biţilor dintr-un **byte**.

Rezultatul afişat este prezentat mai jos:

```
0 1 1 1 1 0 1 1
```

Bucla **for** testează pe rând fiecare bit din **val**, utilizând operaţia ŞI pe biţi, pentru a determina dacă bitul este 1 sau 0. Dacă bitul este 1, se afişează cifra 1; în caz contrar, se afişează cifra 0. În cadrul proiectului 5-3, veţi observa cum se poate extinde această idee pentru a crea o clasă care afişează biţii din orice valoare de tip întreg.

Operaţia SAU pe biţi, fiind complementară lui ŞI, se poate utiliza pentru a face anumiţi biţi egali cu 1. Orice bit 1 în oricare dintre operanzi va determina ca bitul corespunzător al rezultatului să fie tot 1. De exemplu:

```
  1 1 0 1 0 0 1 1
  1 0 1 0 1 0 1 0
| ---------------
  1 1 1 1 1 0 1 1
```

Se poate utiliza operaţia SAU pentru a transforma programul de conversie în litere mari într-un program de conversie în litere mici, după cum urmează:

```
// Conversia in litere mici.
using System;

class LowCase {
  public static void Main() {
    char ch;

    for(int i = 0; i < 10; i++) {
      ch = (char) ('A' + i);
      Console.Write(ch);

      // facem bitul 6 egal cu 1.
      ch = (char) (ch | 32); // ch este acum litera mica
```

```
            Console.Write(ch + " ");
        }
    }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj
```

Programul efectuează un SAU între fiecare caracter şi valoarea 32, care în binar se reprezintă ca 0000 0000 0010 0000, având deci numai bitul al şaselea egal cu 1. La efectuarea unui SAU între această valoare şi orice altă valoare, cel de-al şaselea bit al rezultatului este 1, ceilalţi rămânând nemodificaţi. După cum am precizat, în cazul caracterelor aceasta determină transformarea majusculelor în literele mici echivalente.

Operaţia de SAU exclusiv (pe scurt XOR) dă valoarea 1 unui bit dacă şi numai dacă biţii care sunt comparaţi diferă, după cum se observă şi din exemplul:

```
  0 1 1 1 1 1 1 1
  1 0 1 1 1 0 0 1
^ ---------------
  1 1 0 0 0 1 1 0
```

Operatorul XOR are proprietatea interesantă de a permite codificarea simplă a unui mesaj. Dacă se efectuează un XOR între o valoare X şi o altă valoare Y, iar apoi între rezultat şi Y se efectuează încă o dată XOR, rezultă valoarea iniţială X. Cu alte cuvinte, considerând secvenţa:

```
R1 = X ^ Y;
R2 = R1 ^ Y;
```

R2 are aceeaşi valoare cu X. Rezultatul unei secvenţe de două operaţii XOR cu aceeaşi valoare este în concluzie chiar valoarea iniţială. Puteţi utiliza acest principiu pentru a dezvolta un program simplu de criptare, care efectuează un XOR între caracterele mesajului şi o cheie întreagă, atât la codificare, cât şi la decodificare. La codificare, operaţia XOR se efectuează pentru prima dată, rezultând textul criptat. Pentru decodificare, se efectuează XOR a doua oară, rezultând textul clar. În continuare prezentăm un exemplu simplu care utilizează această metodă pentru codificarea şi decodificarea unui mesaj scurt:

```
// Utilizarea operatiei XOR la codificarea si
// decodificarea unui mesaj.

using System;

class Encode {
  public static void Main() {
    string msg = "Acesta este un test";
    string encmsg = "";
    string decmsg = "";
    int key = 88;
```



```
    Console.Write("Mesajul initial: ");
    Console.WriteLine(msg);

    // codificam mesajul
    for(int i = 0; i < msg.Length; i++)
      encmsg = encmsg + (char) (msg[i] ^ key);

    Console.Write("Mesajul criptat: ");
    Console.WriteLine(encmsg);

    // decodificam mesajul
    for(int i = 0; i < msg.Length; i++)
      decmsg = decmsg + (char) (encmsg[i] ^ key);

    Console.Write("Mesajul decodificat: ");
    Console.WriteLine(decmsg);
  }
}
```

Construirea mesajului criptat.

Construirea mesajului decriptat.

Iată textul afişat de program:

```
Mesajul initial: Acesta este un test
Mesajul criptat: ;=+,9x=+,=x-6x,=+,
Mesajul decodificat: Acesta este un test
```

După cum se observă, rezultatul a două operaţii XOR succesive cu aceeaşi cheie este chiar textul clar.

Operatorul unar „complement faţă de 1" (NON) inversează starea tuturor biţilor operandului. De exemplu, dacă un întreg numit **A** are valoarea binară 1001 0110, atunci **~A** va fi în binar 0110 1001.

Programul următor prezintă utilizarea operatorului NON afişând un număr şi complementul său în formă binară:

```
// Prezinta operatorul NON la nivel de biti.
using System;

class NonDemo {
  public static void Main() {
    sbyte b = -34;

    for(int t = 128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      else Console.Write("0 ");
    }
    Console.WriteLine();

    // inversam toti bitii
    b = (sbyte) ~b;

    for(int t = 128; t > 0; t = t/2) {
      if((b & t) != 0) Console.Write("1 ");
      else Console.Write("0 ");
    }
  }
}
```

Rezultatul afişat este cel de mai jos:

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

Operatori de deplasare

În C#, este posibil să deplasaţi biţii care alcătuiesc o valoare la stânga sau la dreapta cu o cantitate precizată. Cei doi operatori de deplasare definiţi în C# sunt prezentaţi mai jos:

<<	Deplasare la stânga
>>	Deplasare la dreapta

Formele generale ale celor doi operatori sunt prezentate în continuare:

val << nr-biti
val >> nr-biti

Mai sus, *val* este valoarea care se deplasează cu numărul de biţi precizat de către *nr-biti*.

O deplasare la stânga determină ca toţi biţii valorii precizate să fie deplasaţi spre stânga cu o poziţie, în poziţia cea mai din dreapta introducându-se un zero. Deplasarea la dreapta determină ca toţi biţii să fie deplasaţi cu o poziţie spre dreapta. În cazul deplasării spre dreapta a unei valori fără semn, se introduce un zero în poziţia cea mai din stânga. Pentru o valoare cu semn, se păstrează valoarea bitului de semn. Amintiţi-vă că numerele întregi negative se reprezintă dând bitului cel mai semnificativ valoarea 1. Deci, dacă valoarea deplasată este negativă, fiecare deplasare spre dreapta introduce un 1 în partea stângă. Dacă valoarea este însă pozitivă, fiecare deplasare spre dreapta introduce un 0 în stânga.

Atât pentru deplasarea la stânga, cât şi pentru cea la dreapta, biţii deplasaţi în afara reprezentării numărului se pierd. O deplasare nu este deci o rotire şi, în consecinţă, nu există nici o metodă de a determina un bit care a fost deplasat în afara reprezentării.

Programul următor ilustrează în mod grafic efectul operaţiilor de deplasare spre stânga şi spre dreapta. Un întreg primeşte iniţial valoarea 1, ceea ce înseamnă că numai bitul său cel mai puţin semnificativ este 1. Se efectuează apoi o secvenţă de opt deplasări la stânga asupra întregului. După fiecare deplasare, se afişează cei opt biţi mai puţin semnificativi ai valorii întregi. Procesul se repetă apoi în sens invers, valoarea 1 fiind acum în bitul al optulea, iar deplasările efectuându-se spre dreapta.

```
// Prezentarea operatorilor de deplasare << si >>.
using System;

class ShiftDemo {
  public static void Main() {
    int val = 1;
```



```
    for(int i = 0; i < 8; i++) {
      for(int t = 128; t > 0; t = t/2) {
        if((val & t) != 0) Console.Write("1 ");
        else Console.Write("0 ");
      }
      Console.WriteLine();
      val = val << 1;  // deplasare la stanga
    }
    Console.WriteLine();

    val = 128;
    for(int i = 0; i < 8; i++) {
      for(int t = 128; t > 0; t = t/2) {
        if((val & t) != 0) Console.Write("1 ");
        else Console.Write("0 ");
      }
      Console.WriteLine();
      val = val >> 1;  // deplasare la dreapta
    }
  }
}
```

Programul afişează rezultatul de mai jos:

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Sfatul expertului

Întrebare: Deoarece reprezentarea binară este bazată pe puterile lui 2, putem utiliza operatorii de deplasare ca o variantă mai rapidă de înmulţire şi împărţire a unei valori întregi cu 2?

Răspuns: Da. Operatorii de deplasare pe biţi se pot utiliza pentru efectuarea foarte rapidă a înmulţirii şi împărţirii cu 2. O deplasare spre stânga dublează o valoare. O deplasare spre dreapta o înjumătăţeşte. Desigur, aceasta rămâne valabil numai atât timp cât nu deplasăm toţi biţii dincolo de un capăt sau celălalt al reprezentării.

Atribuiri compuse la nivel de biți

Toți operatorii binari la nivel de biți se pot utiliza în atribuiri compuse. De exemplu, ambele instrucțiuni următoare atribuie lui **x** rezultatul operației XOR dintre **x** și valoarea 127:

```
x = x ^ 127;
x ^= 127;
```

Proiectul 5-3: Clasa ShowBits

Acest proiect creează o clasă numită **ShowBits** care vă permite afișarea în binar a unei valori întregi. O astfel de clasă este destul de utilă în programare. De exemplu, când depanăm codul unui driver pentru un dispozitiv, posibilitatea de a urmări fluxul de date în binar reprezintă de multe ori un avantaj prețios.

Pas cu pas

1. Creați un fișier numit **ShowBits.cs**.
2. Clasa **ShowBits** va începe după cum urmează:

```
class ShowBits {
  public int numbits;

  public ShowBits(int n) {
    numbits = n;
  }
```

ShowBits creează un obiect care afișează un număr precizat de biți. De exemplu, pentru a crea un obiect care afișează cei opt biți mai puțin semnficativi ai unei valori întregi, vom scrie:

```
ShowBits b = new ShowBits(8);
```

Numărul biților de afișat este memorat în variabila **numbits**.

3. Pentru afișarea valorii binare, clasa **ShowBits** pune la dispoziție metoda **show()**, prezentată mai jos:

```
public void show(ulong val) {
  ulong mask = 1;

  // deplasam unitatea spre stanga pana in pozitia necesara
  mask <<= numbits - 1;

  int spacer = 0;
  for(; mask != 0; mask >>=1) {
    if((val & mask) != 0) Console.Write("1");
    else Console.Write("0");
    spacer++;
    if((spacer % 8) == 0) {
      Console.Write(" ");
      spacer = 0;
    }
  }
  Console.WriteLine();
}
```


Remarcați că parametrul lui **show()** este de tipul **ulong**. Aceasta nu înseamnă că sunteți obligați să transmiteți de fiecare dată metodei **show()** la apel o valoare **ulong**. Datorită promovărilor automate de tip din C#, puteți transmite metodei **show()** orice valoare de tip întreg. Numărul biților afișați este determinat de valoarea memorată în **numbits**. La fiecare grupă de opt biți, **show()** afișează un spațiu. Vizualizarea valorilor binare ale întregilor lungi devine astfel mai clară.

4. Programul **ShowBitsDemo** este prezentat mai jos:

```
/*
   Proiectul 5-3

   O clasa care afiseaza reprezentarea binara a unei valori.
*/

using System;

class ShowBits {
  public int numbits;

  public ShowBits(int n) {
    numbits = n;
  }

  public void show(ulong val) {
    ulong mask = 1;

    // deplasam unitatea spre stanga pana in pozitia necesara
    mask <<= numbits - 1;

    int spacer = 0;
    for(; mask != 0; mask >>=1) {
      if((val & mask) != 0) Console.Write("1");
      else Console.Write("0");
      spacer++;
      if((spacer % 8) == 0) {
        Console.Write(" ");
        spacer = 0;
      }
    }
    Console.WriteLine();
  }
}

// Utilizam ShowBits.

class ShowBitsDemo {
  public static void Main() {
    ShowBits b = new ShowBits(8);
    ShowBits i = new ShowBits(32);
    ShowBits li = new ShowBits(64);

    Console.WriteLine("123 in binar: ");
    b.show(123);

    Console.WriteLine("\n87987 in binar: ");
    i.show(87987);
```

```
        Console.WriteLine("\n237658768 in binar: ");
        li.show(237658768);

        // putem de asemenea afisa bitii mai putin semnificativi pentru orice intreg
        Console.WriteLine("\nCel mai putin semnificativ octet al lui 87987: ");
        b.show(87987);
    }
}
```

5. Rezultatul programului **ShowBitsDemo** este prezentat mai jos:

```
123 in binar:
01111011

87987 in binar:
00000000 00000001 01010111 10110011

237658768 in binar:
00000000 00000000 00000000 00000000 00001110 00101010 01100010 10010000

Cel mai putin semnificativ octet al lui 87987:
10110011
```

Exerciţii la minut

- Care sunt tipurile care permit aplicarea operatorilor la nivel de biţi?
- Ce efect au operatorii >> şi <<?
- Ce este incorect în secvenţa de mai jos:

```
byte val = 10;
val = val << 2;
```

Operatorul ?

Unul dintre cei mai fascinanţi operatori din C# este ?. Operatorul ? se utilizează adesea pentru a înlocui instrucţiuni **if-else** având forma generală:

if(*condiţie*)
var = expresie1;
else
var = expresie2;

Valoarea atribuită mai sus lui *var* depinde de rezultatul expresiei *condiţie* care controlează instrucţiunea **if**.

Operatorul ? se numeşte *ternar*, deoarece necesită prezenţa a trei operanzi. Forma generală a operatorului este:

- Operatorii la nivel de biţi se pot aplica asupra tipurilor întregi.
- >> efectuează o deplasare la dreapta, iar << efectuează o deplasare la stânga.
- La efectuarea deplasării la stânga, **val** este promovată la **int**, iar rezultatul operaţiei este tot de tip **int**. Pentru a atribui o valoare **int** unei variabile **byte**, trebuie să utilizaţi un cast.

exp1 **?** *exp2* **:** *exp3*;

unde *exp1* este o expresie de tipul **bool**, iar *exp2* şi *exp3* sunt expresii. Expresiile *exp2* şi *exp3* trebuie să fie de acelaşi tip. Remarcaţi cum este utilizat şi locul unde este amplasat caracterul două puncte.

Valoarea unei expresii ? se determină astfel: se evaluează mai întâi *exp1*. Dacă aceasta este adevărată, atunci se evaluează *exp2*, rezultatul acesteia devenind valoarea întregii expresii ?. Dacă însă *exp1* este falsă, atunci se evaluează *exp3*, valoarea acesteia devenind valoarea expresiei. Să considerăm acest exemplu, care atribuie lui **absval** valoarea absolută (modulul) lui **val**:

```
absval = val < 0 ? -val : val;  // determina modulul lui val
```

Dacă **val** este mai mare sau egală cu zero, **absval** primeşte valoarea sa. Dacă însă **val** este negativă, atunci lui **absval** i se va atribui opusul acestei valori (care reprezintă o valoare pozitivă). Acelaşi cod se poate rescrie utilizând o structură **if-else** astfel:

```
if(val < 0) absval = -val;
else absval = val;
```

Iată un alt exemplu cu operatorul ?. Programul efectuează împărţirea a două numere, evitând însă împărţirea la zero:

```
// Evitarea impartirii la zero, utilizand ?
using System;

class NoZeroDiv {
  public static void Main() {
    int result;

    for(int i = -5; i < 6; i++) {
      result = i != 0 ? 100 / i : 0;
      if(i != 0)
        Console.WriteLine("100 / " + i + " este " + result);
    }
  }
}
```

Se evită împărţirea la zero.

Rezultatul afişat de program este prezentat mai jos:

```
100 / -5 este -20
100 / -4 este -25
100 / -3 este -33
100 / -2 este -50
100 / -1 este -100
100 / 1 este 100
100 / 2 este 50
100 / 3 este 33
100 / 4 este 25
100 / 5 este 20
```

Remarcaţi mai ales următoarea linie din program:

```
result = i != 0 ? 100 / i : 0;
```

Variabila **result** primeşte aici rezultatul împărţirii lui 100 la **i**. Această împărţire are însă loc numai dacă **i** nu este zero. Când **i** este nul, se atribuie lui **result** valoarea 0.

Nu este neapărat să atribuiţi valoarea întoarsă de ? unei variabile. De exemplu, puteţi utiliza valoarea ca parametru efectiv la apelul unei metode. Sau, dacă toate cele trei expresii sunt de tipul **bool**, expresia ? poate fi condiţia care controlează o buclă sau o instrucţiune **if**. Spre exemplu, iată o variantă ceva mai eficientă a programului precedent. Aceasta afişează acelaşi text ca şi mai sus.

```
// Evitarea impartirii la zero, utilizand ?
using System;

class NoZeroDiv2 {
  public static void Main() {

    for(int i = -5; i < 6; i++) {
    if(i != 0 ? true : false)
      Console.WriteLine("100 / " + i + " este " + 100 / i);
    }
  }
}
```

Remarcaţi instrucţiunea **if**. Dacă **i** este zero, rezultatul expresiei condiţionale este **false**. În acest caz, se evită împărţirea cu zero şi nu se afişează nimic. În caz contrar, împărţirea este efectuată.

Verificarea cunoştinţelor

1. Arătaţi cum se declară un tablou unidimensional cu 12 elemente de tipul **double**.
2. Declaraţi un tablou bidimensional cu elemente de tipul **int** şi cu dimensiunile 4 x 5.
3. Declaraţi un tablou bidimensional în scară de tipul **int**, în care prima dimensiune este 5.
4. Iniţializaţi un tablou unidimensional de tipul **int** cu numerele de la 1 la 5.
5. Prezentaţi bucla **foreach**. Arătaţi forma generală a instrucţiunii.
6. Scrieţi un program care utilizează un tablou pentru a determina media a 10 valori de tipul **double**. Utilizaţi orice set de valori doriţi.
7. Modificaţi programul de sortare din proiectul 5-1 astfel încât să sorteze un tablou de stringuri. Luaţi un exemplu pentru a demonstra că programul funcţionează.
8. Ce diferenţă este între metodele **IndexOf()** şi **LastIndexOf()** din clasa **string**?
9. Modificaţi clasa **Encode** care realizează criptarea unui mesaj astfel încât să utilizeze un **string** de 8 caractere pe post de cheie de criptare.
10. Se pot utiliza operatorii la nivel de biţi pe valori de tipul **double**?

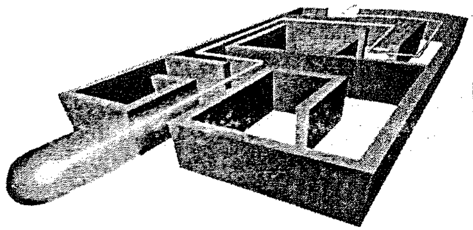
11. Arătaţi cum se poate rescrie secvenţa de mai jos utilizând operatorul ?.

```
if(x < 0) y = 10;
else y = 20;
```

12. În fragmentul următor, **&** reprezintă un operator la nivel de biţi sau un operator logic? De ce?

```
bool a, b;
// ...
if(a & b) ...
```

Modulul 6



O privire mai detaliată asupra metodelor şi claselor

Scopurile acestui modul

- **Controlul accesului la membri**
- **Transmiterea obiectelor către metode**
- **Metode care întorc ca rezultat obiecte**
- **Utilizarea parametrilor `ref` şi `out`**
- **Supraîncărcarea metodelor**
- **Supraîncărcarea constructorilor**
- **Întoarcerea valorilor din metoda `Main()`**
- **Parametrii metodei `Main()`**
- **Utilizarea recursivităţii**
- **Cuvântul cheie `static`**

Acest modul reia discuţia despre clase şi metode. Modulul începe cu explicarea modulului de control al accesului la membrii unei clase. Sunt apoi prezentate transmiterea şi întoarcerea ca rezultat a obiectelor, supraîncărcarea metodelor, diferitele forme ale metodei **`Main()`**, recursivitatea şi utilizarea cuvântului cheie **`static`**.

Controlul accesului la membrii unei clase

Ca urmare a faptului că reprezintă o implementare a conceptului de încapsulare, clasele oferă două avantaje majore. Mai întâi, ele combină datele cu codul care le prelucrează. Aţi beneficiat deja de acest avantaj oferit de clase încă din modulul 4. În al doilea rând, clasele oferă mijloacele prin care accesul la membrii lor poate fi controlat. Această facilitate este examinată în continuare.

Deşi abordarea acestei probleme în C# este ceva mai complicată, în general, există două categorii fundamentale de membri ai unei clase: membri publici şi membri privaţi. Membrii *publici* pot fi accesaţi în mod liber de codul definit în afara clasei. Acesta este tipul de membri pe care l-am utilizat până acum. Membrii *privaţi*, în schimb, sunt accesibili numai metodelor definite în aceeaşi clasă. Utilizarea membrilor privaţi implementează de fapt controlul accesului la membrii clasei.

Restrângerea accesului la membrii unei clase reprezintă o parte fundamentală a programării orientate spre obiecte, deoarece previne utilizarea eronată a obiectelor. Dacă permiteţi accesul la datele private numai prin intermediul unui set bine definit de metode, puteţi preveni atribuirea unor valori improprii acelor date – efectuând de exemplu o verificare de domeniu. Metodele din afara clasei nu pot modifica direct valoarea unui membru privat al clasei. Puteţi de asemenea controla în mod exact cum şi când sunt utilizate datele din cadrul unui obiect. Atunci când este corect implementată, o clasă creează deci o „cutie neagră" care poate fi utilizată, dar ale cărei detalii interne sunt ferite de modificarea din afară.

Specificatorii de acces din C#

Controlul accesului la membrii unei clase se realizează prin utilizarea a patru specificatori de acces: **`public`**, **`private`**, **`protected`** şi **`internal`**. În acest modul, ne vom ocupa doar de primii doi: **`public`** şi **`private`**. Modificatorul **`protected`** este legat de conceptul de moştenire şi va fi descris în modulul 8. Modificatorul **`internal`** se referă în majoritatea cazurilor la utilizarea unui *grup*, care pentru C# înseamnă un program, un proiect sau o componentă. Modificatorul **`internal`** va fi prezentat pe scurt în modulul 12.

Dacă un membru al unei clase se găseşte sub influenţa specificatorului **`public`**, membrul este accesibil în orice altă parte din program. Aceasta include şi metodele definite în cadrul altor clase.

Dacă un membru al unei clase este specificat ca fiind **`private`**, acel membru poate fi accesat numai de ceilalţi membri ai clasei sale. Metodele altor clase nu vor putea prin urmare să aibă acces la membrul **`private`**. După cum s-a precizat în modulul 4, dacă specificatorul de acces lipseşte, un membru al unei clase este implicit privat. Specificatorul **`private`** nu este deci obligatoriu la crearea membrilor privaţi ai clasei.

Specificatorul de acces precede restul descrierii de tip a unui membru dintr-o clasă. Cu alte cuvinte, specificatorul apare pe prima poziţie în instrucţiunea de declaraţie a membrului. Mai jos sunt câteva exemple:

```
public string errMsg;
private double bal;
private bool isError(byte status) { // ...
```

Pentru a înţelege diferenţa dintre **`public`** şi **`private`**, să considerăm următorul program:

```
// Accesul public sau privat.

using System;

class MyClass {
  private int alfa;
  int beta;
  public int gama;
```

```
/* Metode care permit accesul la alfa si beta. Este permis ca un membru
   al unei clase sa utilizeze un membru privat din aceeasi clasa.
*/
public void setAlfa(int a) {
  alfa = a;
}

public int getAlfa() {
  return alfa;
}

public void setBeta(int a) {
  beta = a;
}

public int getBeta() {
  return beta;
}
}

class AccessDemo {
  public static void Main() {
    MyClass ob = new MyClass();

    /* Accesul la alfa si beta este permis numai prin intermediul metodelor. */
    ob.setAlfa(-99);
    ob.setBeta(19);
    Console.WriteLine("ob.alfa este " + ob.getAlfa());
    Console.WriteLine("ob.beta este " + ob.getBeta());

    // Nu puteti accesa alfa sau beta astfel:
//  ob.alfa = 10; // Incorect! alfa este privat!
//  ob.beta = 9;  // Incorect! beta este privat!

    // Este permis sa accesati direct gama, deoarece este public.
    ob.gama = 99;
  }
}
```

Greşit, **alfa** şi **beta** sunt **private** !

Corect, **gama** fiind **public**.

După cum observaţi, în interiorul clasei **MyClass**, **alfa** este specificat ca fiind **private**, **beta** este privat în mod implicit, iar **gama** este specificat ca **public**. Deoarece **alfa** şi **beta** sunt private, ele nu sunt accesibile codului din afara clasei lor. În consecinţă, nici una dintre ele nu poate fi utilizată direct în clasa **AccessDemo**. Ele trebuie să fi accesate prin intermediul unor metode publice, cum sunt **setAlfa()** şi **getAlfa()**. De exemplu, dacă încercaţi să ştergeţi semnul de comentariu de la începutul liniei următoare:

```
// ob.alfa = 10; // Incorect! alfa este privat!
```

nu veţi mai putea compila programul, din cauza violării accesului. Deşi accesul codului din afara clasei **MyClass** la **alfa** este interzis, metodele definite în clasa **MyClass** o pot utiliza în mod liber, după cum arată şi metodele **setAlfa()** şi **getAlfa()**. Acelaşi lucru rămâne valabil şi pentru **beta**.

Ideea de bază este: membrii privaţi pot fi utilizaţi în mod liber de ceilalţi membri din aceeaşi clasă, dar sunt inaccesibili pentru codul din afara clasei lor.

Pentru a observa cum se aplică regulile de control al accesului într-un exemplu practic, să considerăm programul următor, care implementează un tablou de tipul **int** în care erorile de depăşire a marginii sunt prevenite, evitându-se producerea excepţiilor la execuţie. Aceasta se realizează prin încapsularea tabloului ca membru privat într-o clasă, accesul la tablou fiind permis numai metodelor membre. Această soluţie previne orice încercare de acces la tablou dincolo de marginile acestuia, eşuarea producându-se elegant (şi aducând mai degrabă cu o „aterizare pe sol moale" decât cu o prăbuşire). Tabloul este implementat de către clasa **FailSoftArray**, prezentată mai jos:

```
/* Aceasta clasa implementeaza un tablou care previne erorile la executie.
 */

using System;

class FailSoftArray {
  private int [] a;     // referinta la tablou
  private int errval;   // valoarea intoarsa cand get() esueaza

  public int Length;    // lungimea este publica

  /* Construim tabloul, cunoscand lungimea sa si
     valoarea intoarsa cand get() esueaza. */

  public FailSoftArray(int size, int errv) {
    a = new int[size];
    errval = errv;
    Length = size;
  }

  // Intoarce valoarea care corespunde cu un index dat.
  public int get(int index) {
    if(ok(index)) return a[index];
    return errval;
  }

  // Scrie o valoare la un anumit index. Intoarce succes sau esec.
  public bool put(int index, int val) {
    if(ok(index)) {
      a[index] = val;
      return true;
    }
    return false;
  }

  // Intoarce true daca index este in domeniul permis de valori
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
  }
}
```

Variabile instanţă private.

Detectează un index în afara domeniului.

O metodă privată.

```
// Utilizeaza un tablou "tolerant la erori".
class FSDemo {
  public static void Main() {
    FailSoftArray fs = new FailSoftArray(5, -1);
    int x;

    // prezentare "silentioasa" a erorilor
    Console.WriteLine("Esuare silentioasa.");
    for(int i = 0; i < (fs.Length * 2); i++)
      fs.put(i, i * 10);

    for(int i = 0; i < (fs.Length * 2); i++) {
      x = fs.get(i);
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine("");

    // tratarea efectiva a erorilor
    Console.WriteLine("\nEsuare cu raportarea erorilor.");
    for(int i = 0; i < (fs.Length * 2); i++)
      if(!fs.put(i, i * 10))
        Console.WriteLine("Indexul " + i + " depaseste marginile");

    for(int i = 0; i < (fs.Length * 2); i++) {
      x = fs.get(i);
      if(x != -1) Console.Write(x + " ");
      else
        Console.WriteLine("Indexul " + i + " depaseste marginile");
    }
  }
}
```

Programul afişează rezultatul prezentat mai jos:

```
Esuare silentioasa.
0 10 20 30 40

Esuare cu raportarea erorilor.
Indexul 5 depaseste marginile
Indexul 6 depaseste marginile
Indexul 7 depaseste marginile
Indexul 8 depaseste marginile
Indexul 9 depaseste marginile
0 10 20 30 40 Indexul 5 depaseste marginile
Indexul 6 depaseste marginile
Indexul 7 depaseste marginile
Indexul 8 depaseste marginile
Indexul 9 depaseste marginile
```

Să examinăm mai amănunţit acest exemplu. În clasa **FailSoftArray** se definesc trei membri privaţi. Primul dintre ei este **a**, care reprezintă o referinţă la tabloul care memorează efectiv informaţia. Cel de-al doilea este **errval**, a cărui valoare va fi întoarsă ori de câte ori apelul lui **get()** eşuează. Al treilea este metoda privată **ok()**, care stabileşte dacă un index a depăşit sau nu marginile. Aceşti trei membri pot fi deci utilizaţi numai de către ceilalţi membri ai clasei **FailSoftArray**. Mai exact,

variabilele **a** şi **errval** vor fi utilizate numai de metodele clasei, iar **ok()** va fi apelată numai de către alţi membri ai clasei **FailSoftArray**. Restul membrilor clasei sunt publici şi deci pot fi utilizaţi în orice alt cod dintr-un program care utilizează clasa **FailSoftArray**.

Când construiţi un obiect de tipul **FailSoftArray**, trebuie să precizaţi dimensiunea tabloului şi valoarea care va fi întoarsă când **get()** eşuează. Aceasta trebuie să fie o valoare care în mod normal nu poate apărea în tablou. După ce a fost construit, tabloul referit efectiv de **a** şi valoarea memorată în **errval** nu pot fi accesate de utilizatorii obiectului **FailSoftArray**. Astfel, ele sunt ferite de utilizarea eronată. De exemplu, utilizatorul nu mai are voie să indexeze direct tabloul **a**, ceea ce putea conduce la depăşirea marginilor. Accesul la tablou este permis numai prin intermediul metodelor **get()** şi **put()**.

Metoda **ok()** este declarată **private** mai ales în scop demonstrativ. Ar fi fost inofensiv să o declarăm **public**, deoarece oricum ea nu modifică obiectul. Fiind însă utilizată numai intern în cadrul clasei **FailSoftArray**, poate fi **private**.

Remarcaţi că variabila instanţă **Length** este **public**. Aceasta asigură compatibilitatea cu maniera în care limbajul C# implementează tablourile. Pentru a obţine lungimea unui obiect **FailSoftArray**, este suficient să utilizaţi membrul **Length**.

Pentru a utiliza un tablou **FailSoftArray**, apelaţi **put()** pentru inserarea unei valori la un index precizat. Apelaţi **get()** pentru a determina valoarea care se găseşte la un index dat. Dacă indexul depăşeşte marginile tabloului, **put()** va întoarce **false**, iar **get()** va întoarce **errval**.

Deoarece membrii unei clase sunt implicit privaţi, nu are nici un rost să-i declarăm utilizând în mod explicit cuvântul cheie **private**. În consecinţă, începând din acest moment, nu vom mai declara în mod redundant membrii claselor ca fiind **private**. Reţineţi însă că, dacă un membru al unei clase nu este precedat de nici un modificator de acces, atunci este privat.

Sfatul expertului

Întrebare: Chiar dacă este adevărat că tablourile „tolerante la erori" din exemplul anterior previn depăşirea marginilor, preţul plătit este imposibilitatea de a utiliza sintaxa obişnuită de indexare a tablourilor. Există vreo modalitate mai bună de a crea un astfel de tablou?

Răspuns: Da. După cum veţi vedea în modulul 7, limbajul C# a prevăzut o categorie specială de entităţi membre ale unei clase denumite *indexări*, care vă permit să indexaţi un obiect la fel cum indexaţi un tablou. Există de asemenea o modalitate mai bună de a memora câmpul **Length**, transformându-l într-o *proprietate*. Şi această facilitate este descrisă tot în modulul 7.

Exerciții la minut

- Precizați specificatorii de acces din C#.
- Explicați ce semnifică **`private`** și **`public`**.
- Care este tipul implicit de acces la membrul unei clase?

Proiectul 6-1: O versiune îmbunătățită a clasei Queue

Beneficiind de accesul privat, puteți acum aduce o îmbunătățire destul de importantă clasei **`Queue`** dezvoltate în cadrul proiectului 5-2 din modulul 5. În acea versiune, toți membrii clasei **`Queue`** erau publici. Aceasta înseamnă că un program care utilizează clasa **`Queue`** poate avea acces direct la tabloul cu care este implementată coada, putând astfel să acceseze elemente peste rând. Deoarece ideea structurii de coadă este de a implementa disciplina „primul sosit, primul servit", tolerarea accesului în afara rândului nu trebuie acceptată. De asemenea, un programator rău intenționat poate modifica direct valorile memorate de indicii **`putloc`** și **`getloc`**, dând astfel peste cap mecanismul de coadă. Din fericire, aceste tipuri de probleme pot fi prevenite cu ușurință prin declararea unor părți ale clasei **`Queue`** ca **`private`**.

Pas cu pas

1. Copiați clasa **`Queue`** din proiectul 5-2 într-un fișier nou, denumit **`Queue.cs`**.
2. În clasa **`Queue`**, eliminați specificatorul **`public`** din fața tabloului **`q`** și a indicilor **`putloc`** și **`getloc`**, ca mai jos:

```
// O coada de caractere imbunatatita
class Queue {
  // acesti membri sunt acum privati
  char [] q; // tabloul care implementeaza coada
  int putloc, getloc; // indicii put si get

  public Queue(int size) {
    q = new char[size + 1]; // se aloca memorie pentru coada
    putloc = getloc = 0;
  }

  // insereaza un caracter in coada
  public void put(char ch) {
    if(putloc == q.Length - 1) {
      Console.WriteLine(" - Coada este plina.");
      return;
    }

    putloc++;
    q[putloc] = ch;
  }
```

- Specificatorii de acces din C# sunt **`private`**, **`public`**, **`protected`** și **`internal`**.
- Un membru **`private`** nu poate fi accesat decât de ceilalți membri ai clasei sale. Un membru **`public`** este accesibil în tot codul programului.
- Orice membru al unei clase este în mod implicit privat.


```
  // intoarce un caracter din coada
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine(" - Coada este vida.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
```

3. Modificarea tipului de acces (de la public la privat) pentru **`q`**, **`putloc`** și **`getloc`** nu influențează în nici un fel un program care utilizează corect clasa **`Queue`**. De exemplu, clasa **`QDemo`** din cadrul proiectului 5-2 continuă să funcționeze corect. Această versiune împiedică însă utilizarea incorectă a obiectelor **`Queue`**. De exemplu, următoarele tipuri de instrucțiuni sunt acum nepermise:

```
Queue test = new Queue(10);

test.q[0] = 'X'; // gresit !
test.putloc = -100; // nu mai functioneaza
```

4. Deoarece membrii **`q`**, **`putloc`** și **`getloc`** sunt privați, clasa **`Queue`** va impune acum cu strictețe disciplina „primul sosit, primul servit" caracteristică oricărei cozi.

Transmiterea obiectelor către metode

Până în prezent, exemplele din această carte au utilizat tipuri valorice, cum ar fi **`int`** sau **`double`**, ca parametri pentru metode. Transmiterea obiectelor ca parametri pentru metode este însă o practică obișnuită și în același timp corectă. De exemplu, să considerăm următorul program care memorează dimensiunile unui paralelipiped dreptunghic:

```
// Obiectele pot fi transmise ca parametri metodelor.

using System;

class Paralelipiped {
  int a, b, c;
  int volum;

  public Paralelipiped(int i, int j, int k) {
    a = i;
    b = j;
    c = k;
    volum = a * b * c;
  }

  // intoarce true daca ob defineste un paralelipiped congruent.
  public bool Congruent(Paralelipiped ob) {
    if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
    else return false;
  }
  // intoarce true daca ob are acelasi volum.
  public bool Echivalent(Paralelipiped ob) {
```

Parametrii sunt obiecte.

```
        if(ob.volum == volum) return true;
        else return false;
    }
}

class TransOb {
    public static void Main() {
        Paralelipiped ob1 = new Paralelipiped(10, 2, 5);
        Paralelipiped ob2 = new Paralelipiped(10, 2, 5);
        Paralelipiped ob3 = new Paralelipiped(4, 5, 5);

        Console.WriteLine("ob1 este congruent cu ob2: " +
                          ob1.Congruent(ob2));
        Console.WriteLine("ob1 este congruent cu ob3: " +
                          ob1.Congruent(ob3));
        Console.WriteLine("ob1 este echivalent cu ob3: " +
                          ob1.Echivalent(ob3));
    }
}
```

Se transmit obiecte.

Programul va afişa următorul text:

```
ob1 este congruent cu ob2: true
ob1 este congruent cu ob3: false
ob1 este echivalent cu ob3: true
```

Metodele `Congruent()` şi `Echivalent()` compară obiectul pentru care sunt invocate cu obiectul primit ca parametru. Metoda `Congruent()` compară toate cele trei dimensiuni, întorcând `true` numai când cele două paralelipipede sunt congruente. Metoda `Echivalent()` compară cele două paralelipipede numai din punctul de vedere al volumelor. În ambele cazuri, remarcaţi că parametrul formal `ob` are `Paralelipiped` specificat ca tip. Acest exemplu demonstrează că, din punct de vedere sintactic, tipurile obiect sunt transmise metodelor la fel ca şi tipurile valorice.

Modul de transfer al parametrilor

După cum s-a demonstrat în exemplul precedent, transmiterea unui obiect ca parametru pentru o metodă este o operaţie simplă. Există însă câteva nuanţe pe care exemplul respectiv nu le surprinde. În anumite cazuri, transmiterea obiectelor determină efecte diferite faţă de cele care se produc la transmiterea tipurilor valorice. Pentru a vedea de ce, trebuie să înţelegeţi cele două modalităţi în care un parametru efectiv poate fi transmis unui subprogram.

Prima dintre aceste modalităţi este *transferul prin valoare.* Metoda copiază *valoarea* parametrului efectiv în parametrul formal al subrutinei. Modificările efectuate asupra parametrului subrutinei nu vor afecta deci valoarea parametrului efectiv utilizat la apel. Cea de-a doua modalitate de transmitere a unui parametru efectiv este *transferul prin referinţă.* Această metodă transmite parametrului formal o *referinţă* către parametrul efectiv (şi nu valoarea acestuia). În interiorul subrutinei, referinţa este utilizată pentru accesul la parametrul efectiv specificat la apel. Aceasta înseamnă că modifi-

cările parametrului formal vor afecta şi parametrul efectiv utilizat la apelul subrutinei. După cum veţi vedea, în C# se utilizează ambele metode de transfer.

În C#, la transmiterea unui tip valoric, cum ar fi `int` sau `double`, către o metodă, transferul se face prin valoare. În consecinţă, ceea ce se întâmplă cu parametrul formal care primeşte valoarea parametrului efectiv nu rămâne vizibil în afara metodei. De exemplu, să considerăm următorul program:

```
// Tipurile simple sunt transferate prin valoare.

using System;

class Test {
    /* Aceasta metoda nu modifica parametrii efectivi utilizati la apel. */
    public void noChange(int i, int j) {
        i = i + j;
        j = -j;
    }
}

class TransfVal {
    public static void Main() {
        Test ob = new Test();
        int a = 15, b = 20;

        Console.WriteLine("a si b inainte de apel: " + a + " " + b);

        ob.noChange(a, b);

        Console.WriteLine("a si b dupa apel: " + a + " " + b);
    }
}
```

Rezultatul afişat de acest program este prezentat mai jos:

```
a si b inainte de apel: 15 20
a si b dupa apel: 15 20
```

După cum se observă, operaţiile care au loc în corpul metodei `noChange()` nu afectează valorile variabilelor `a` şi `b`, utilizate la apel.

Atunci când transmiteţi o referinţă către un obiect unei metode, situaţia este ceva mai complicată. Din punct de vedere tehnic, referinţa însăşi este transferată prin valoare. În consecinţă, se face o copie a referinţei, iar modificările asupra parametrului formal nu afectează parametrul efectiv. (De exemplu, dacă facem ca parametrul formal să refere un alt obiect, parametrul efectiv va referi în continuare obiectul iniţial.) Ceea ce este însă foarte important este că *modificările asupra obiectului* referit de parametrul formal *vor afecta* obiectul referit de parametrul efectiv. Să justificăm.

Să ne amintim că la crearea unei variabile al cărei tip este o clasă, se creează o referinţă către un obiect şi nu un obiect propriu-zis. Obiectul este alocat de `new`, iar o referinţă către el este atribuită unei variabilei referinţă. Atunci când utilizăm variabila referinţă ca parametru efectiv al unei metode, parametrul formal primeşte o referinţă către obiectul pe care îl referă şi parametrul efectiv. Aşadar, parametrul

efectiv şi cel formal referă ambele acelaşi obiect. Aceasta explică de fapt mecanismul de transfer al obiectelor prin referinţă. Modificările asupra obiectului din corpul unei metode *afectează* obiectul transmis ca parametru efectiv. Spre exemplu, să considerăm următorul program:

```
// Obiectele sunt transferate implicit prin referinta.

using System;

class Test {
  public int a, b;

  public Test(int i, int j) {
    a = i;
    b = j;
  }

  /* Transmite un obiect. Acum, ob.a si ob.b din apel vor fi modificate. */
  public void change(Test ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}

class TransfRef {
  public static void Main() {
    Test ob = new Test(15, 20);

    Console.WriteLine("ob.a si ob.b inainte de apel: " + ob.a + " " + ob.b);

    ob.change(ob);

    Console.WriteLine("ob.a si ob.b dupa apel: " + ob.a + " " + ob.b);
  }
}
```

Acest program va genera următorul text:

```
ob.a si ob.b inainte de apel: 15 20
ob.a si ob.b dupa apel: 35 -20
```

După cum puteţi observa, în acest caz modificările din corpul metodei **change()** au afectat obiectul utilizat ca parametru efectiv.

Să recapitulăm: la transmiterea către o metodă a unei referinţe către un obiect, referinţa se transferă prin valoare. Se face deci o copie a referinţei. Deoarece valoarea transferată referă un obiect, copia acelei valori va referi acelaşi obiect ca şi parametrul efectiv corespunzător.

Utilizarea parametrilor ref şi out

După cum am explicat, în mod implicit tipurile valorice, cum ar fi **int** sau **char**, sunt transferate către metode prin valoare. Aceasta înseamnă că modificările asupra parametrului formal care primeşte un tip valoric nu afectează parametrul efectiv

utilizat la apel. Este însă posibil să schimbaţi acest comportament. Utilizând cuvintele cheie **ref** şi **out**, este posibil să transferaţi oricare dintre tipurile valorice prin referinţă. Aceasta va permite unei metode să modifice parametrii efectivi utilizaţi la apel.

Înainte de detaliile tehnice ale utilizării lui **ref** şi **out**, este util să înţelegeţi de ce este uneori necesar să transferaţi un tip valoric prin referinţă. În general, sunt două motive: pentru a permite unei metode să modifice conţinutul parametrilor săi efectivi sau pentru a permite unei metode să întoarcă mai mult de o valoare. Să studiem fiecare dintre aceste motive în detaliu.

Uneori, aveţi nevoie ca o metodă să poată modifica parametrii efectivi pe care îi primeşte. Exemplul de bază pentru această situaţie este cel al metodei **swap()**, care interschimbă valorile celor doi parametri pe care îi primeşte. Deoarece tipurile de bază sunt transferate prin valoare, nu este posibil să scrieţi o astfel de metodă care interschimbă două valori de tipul **int**, de exemplu, utilizând mecanismul implicit de transfer prin valoare din C#. Această problemă este rezolvată de modificatorul **ref**.

După cum cunoaşteţi, o instrucţiune **return** permite unei metode să întoarcă o valoare modulului apelant. O metodă poate însă întoarce numai *o singură valoare* de fiecare dată când este apelată. Cum procedaţi dacă aveţi nevoie să întoarceţi două sau mai multe informaţii? De exemplu, cum faceţi să creaţi o metodă care calculează aria unui dreptunghi şi în acelaşi timp determină dacă dreptunghiul este pătrat? Aceasta presupune întoarcerea a două informaţii: aria dreptunghiului şi o valoare care indică dacă acesta este sau nu pătrat. Această metodă nu poate fi scrisă utilizând o singură valoare de întoarcere. Problema este rezolvată cu ajutorul modificatorului **out**.

Utilizarea lui ref

Modificatorul de parametri **ref** forţează în C# transferul prin referinţă în detrimentul transferului prin valoare. Modificatorul **ref** apare atât la declaraţia, cât şi la apelul unei metode. Vom începe cu un exemplu foarte simplu. Programul următor creează o metodă numită **sqr()** care îşi înlocuieşte parametrul întreg transmis cu pătratul acestuia. Remarcaţi utilizarea şi dispunerea lui **ref**.

```
// Utilizarea lui ref pentru a forta transferul tipurilor
// valorice prin referinta.

using System;

class RefTest {
  // Aceasta metoda isi modifica parametrul.
  public void sqr(ref int i) {   <-- ref precede aici declaratia parametrului.
    i = i * i;
  }
}

class RefDemo {
  public static void Main() {
    RefTest ob = new RefTest();
```

```
    int a = 10;

    Console.WriteLine("a inainte de apel: " + a);

    ob.sqr(ref a);

    Console.WriteLine("a dupa apel: " + a);
  }
}
```

Aici, **ref** stă în fața parametrului efectiv.

Remarcați că **ref** precede întreaga declarație a parametrului formal al metodei și de asemenea precede numele parametrului efectiv la apelul metodei. Rezultatul acestui program, prezentat mai jos, confirmă faptul că valoarea parametrului efectiv **a** este modificată în corpul metodei **sqr()**:

```
a inainte de apel: 10
a dupa apel: 100
```

Utilizând **ref**, este acum posibil să scriem o metodă care interschimbă valorile a doi parametri de tipuri valorice. De exemplu, programul din continuare conține o metodă numită **swap()** care interschimbă valorile a doi parametri întregi atunci când este apelată:

```
// Interschimba doua valori.

using System;

class Swap {
  // Aceasta metoda interschimba valorile parametrilor.
  public void swap(ref int a, ref int b) {
    int t;

    t = a;
    a = b;
    b = t;
  }
}

class SwapDemo {
  public static void Main() {
    Swap ob = new Swap();

    int x = 10, y = 20;

    Console.WriteLine("x si y inainte de apel: " + x + " " + y);

    ob.swap(ref x, ref y);

    Console.WriteLine("x si y dupa apel: " + x + " " + y);
  }
}
```

Rezultatul afișat de acest program este prezentat mai jos:

```
x si y inainte de apel: 10 20
x si y dupa apel: 20 10
```


Mai este un aspect important de reținut privitor la **ref**: orice parametru transferat cu **ref** trebuie să aibă o valoare atribuită înainte de apel. Motivul este că metoda care primește un astfel de parametru presupune că acesta referă o valoare corectă. Prin urmare, nu puteți scrie o metodă care să inițializeze un parametru cu ajutorul lui **ref**.

Utilizarea lui out

Uneori, aveți nevoie să utilizați un parametru transmis prin referință pentru a primi o valoare în corpul unei metode, fără ca parametrul să aibă o valoare inițială la apel. Spre exemplu, o metodă poate efectua o anumită operație, cum ar fi deschiderea unui socket în rețea, care întoarce un cod de stare într-un parametru referință. În acest caz, nu este necesar să transmitem nici o informație metodei, de la care trebuie însă să primim înapoi o informație. Problema acestui scenariu este că parametrii **ref** trebuie inițializați cu o valoare înainte de apel. Utilizarea unui parametru **ref** v-ar forța deci să dați parametrului o valoare nesemnificativă doar pentru a satisface această restricție. Din fericire, limbajul C# pune la dispoziție o soluție mai bună: parametrii **out**.

Un parametru **out** este similar cu un parametru **ref**, cu o singură diferență: poate fi utilizat numai pentru a întoarce o valoare dintr-o metodă. Nu este necesar (și nici util) să inițializați variabila utilizată ca parametru **out** cu o valoare înaintea apelului metodei. Metoda va da acestei variabile o valoare. Mai mult, în corpul metodei, un parametru **out** este întotdeauna considerat ca fiind *neinițializat*, adică se presupune că nu are o valoare inițială. Metoda *trebuie* să atribuie acestui parametru o valoare înainte de a se termina. În concluzie, după apelul unei metode, un parametru **out** va conține întotdeauna o valoare.

În continuare, prezentăm un exemplu care utilizează un parametru **out**. Metoda **rectInfo()** întoarce aria unui dreptunghi cunoscând lungimile laturilor acestuia. În parametrul **estePatrat**, metoda întoarce **true** dacă dreptunghiul este pătrat și **false** în caz contrar. În concluzie, **rectInfo()** va întoarce două informații în modulul apelant.

```
// Utilizarea unui parametru out.

using System;

class Rectangle {
  int lat1;
  int lat2;

  public Rectangle(int i, int j) {
    lat1 = i;
    lat2 = j;
  }

  // intoarce aria si determina daca este patrat.
  public int rectInfo(out bool estePatrat) {
```

Întoarce o informație utilizând un parametru **out**.

```
    if(lat1 == lat2) estePatrat = true;
    else estePatrat = false;
    return lat1 * lat2;
  }
}

class RectDemo {
  public static void Main() {
    Rectangle rect = new Rectangle(10, 23);
    int aria;
    bool estePtr;

    aria = rect.rectInfo(out estePtr);

    if(estePtr) Console.WriteLine("rect este patrat.");
    else Console.WriteLine("rect nu este patrat.");

    Console.WriteLine("Aria lui rect este " + aria + ".");
  }
}
```

Remarcați că **estePtr** nu primește o valoare inițială înaintea apelului metodei **rectInfo()**. Aceasta nu ar fi fost permis dacă parametrul lui **rectInfo()** ar fi fost **ref** și nu **out**. După revenirea din metodă, **estePtr** va conține **true** sau **false**, după cum dreptunghiul este sau nu pătrat. Aria dreptunghiului este întoarsă prin intermediul instrucțiunii **return**. Rezultatul programului este prezentat mai jos:

```
rect nu este patrat.
Aria lui rect este 230.
```

Exerciții la minut

- Ce diferență este între transferul prin valoare și transferul prin referință?
- Cum se transferă tipurile valorice în C#? Dar obiectele?
- Ce rol are **ref**? Care este diferența între **ref** și **out**?

- La transferul prin valoare, o copie a parametrului este transferată subrutinei. În cazul transferului prin referință, se transmite o referință către parametru.
- Tipurile valorice sunt transferate prin valoare, iar obiectele prin referință.
- Modificatorul **ref** creează posibilitatea transferului prin referință pentru tipurile valorice. Modificatorul **out** forțează de asemenea transferul prin referință, dar poate fi utilizat numai pentru întoarcerea unei valori dintr-o metodă.

Sfatul expertului

Întrebare: Se pot utiliza **ref** și **out** și asupra tipurilor referință, de exemplu la transferul unei referințe către un obiect?

Răspuns: Da. Atunci când **ref** sau **out** afectează un parametru de tip referință, referința însăși va fi transferată prin referință. Aceasta permite unei metode să modifice obiectul referit de către un anumit parametru referință. Să considerăm următorul program:

```
// Utilizam ref pentru un parametru de tip obiect.

using System;

class Test {
  public int a;

  public Test(int i) {
    a = i;
  }
  // Metoda care nu isi modifica parametrul.
  public void noChange(Test o) {
    Test newob = new Test(0);
    o = newob;  // nu are efect in afara lui noChange();
  }

  // Metoda care schimba obiectul referit de parametru.
  public void change(ref Test o) {
    Test newob = new Test(0);
    o = newob(); // modifica parametrul efectiv.
  }
}

class TransfObCuRef {
  public static void Main() {
    Test ob = new Test(100);

    Console.WriteLine("ob.a inainte de apel: " + ob.a);

    ob.noChange(ob);
    Console.WriteLine("ob.a dupa apelul lui noChange(): " + ob.a);

    ob.change(ref ob);
    Console.WriteLine("ob.a dupa apelul lui change(): " + ob.a);
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
ob.a inainte de apel: 100
ob.a dupa apelul lui noChange(): 100
ob.a dupa apelul lui change(): 0
```

După cum se poate observa, atunci când lui **o** i se atribuie o referință către un nou obiect în corpul metodei **noChange()**, parametrul efectiv **ob** din **Main()** nu este afectat. În corpul metodei **change()**, care utilizează un parametru **ref**, atribuirea unui alt obiect lui **o** duce la modificarea obiectului referit de **ob** în **Main()**.

Utilizarea unui număr variabil de parametri

La crearea unei metode, numărul de parametri transmişi metodei este de obicei cunoscut dinainte, dar nu întotdeauna se întâmplă astfel. Uneori puteţi avea nevoie de o metodă care poate primi un număr arbitrar de parametri. De exemplu, să considerăm o metodă care determină minimul unei secvenţe de valori. O astfel de metodă poate primi numai două valori, dar şi trei, sau patru ş.a.m.d. În toate situaţiile, această metodă trebuie să întoarcă valoarea minimă. O astfel de metodă nu se poate crea utilizând o secvenţă obişnuită de parametri. Trebuie să recurgeţi în schimb la un tip special de parametru care poate înlocui un număr arbitrar de parametri. Aceasta se realizează cu ajutorul unui parametru de tipul **params**.

Modificatorul **params** se utilizează pentru declararea unui tablou de parametri, care poate memora zero sau mai mulţi parametri. Numărul elementelor din tablou va corespunde cu numărul parametrilor efectivi transmişi metodei. Programul dumneavoastră va accesa apoi acest tablou pentru a-şi obţine parametrii efectivi.

Mai jos, prezentăm un exemplu care utilizează **params** pentru crearea unei metode numite **minVal()**, care întoarce valoarea minimă dintr-o secvenţă de valori:

```
// Prezentarea params.

using System;

class Min {
  public int minVal(params int[] nums) {   <-- Creează un parametru de lungime variabilă cu params.
    int m;

    if(nums.Length == 0) {
      Console.WriteLine("Eroare: nu sunt parametri.");
      return 0;
    }

    m = nums[0];
    for(int i = 1; i < nums.Length; i++)
      if(nums[i] < m) m = nums[i];

    return m;
  }
}

class ParamsDemo {
  public static void Main() {
    Min ob = new Min();
    int min;
    int a = 10, b = 20;

    // apel cu doua valori
    min = ob.minVal(a, b);
    Console.WriteLine("Minimul este " + min);

    // apel cu 3 valori
    min = ob.minVal(a, b, -1);
    Console.WriteLine("Minimul este " + min);

    // apel cu 5 valori
    min = ob.minVal(18, 23, 3, 14, 25);
    Console.WriteLine("Minimul este " + min);

    // apelul cu un tablou de intregi este corect
    int [] args = { 45, 67, 34, 9, 112, 8 };
    min = ob.minVal(args);
    Console.WriteLine("Minimul este " + min);
  }
}
```


Rezultatul afişat de program este prezentat mai jos:

```
Minimul este 10
Minimul este -1
Minimul este 3
Minimul este 8
```

La fiecare apel al metodei **minVal()**, parametrii îi sunt transmişi prin intermediul tabloului **nums**. Lungimea acestui tablou este egală cu numărul elementelor din secvenţă. Puteţi deci utiliza **minVal()** pentru a determina minimul unui număr arbitrar de valori.

Chiar dacă un parametru **params** poate acoperi orice număr de parametri, toţi aceştia trebuie să fie de un tip compatibil cu tipul tabloului specificat. De exemplu, apelul următor al metodei **minVal()**:

```
min = ob.minVal(1, 2.2);
```

este incorect, deoarece nu există o conversie automată de la **double** (2.2) la **int**, care este tipul tabloului **nums** din **minVal()**.

Atunci când utiizaţi **params**, trebuie să aveţi grijă la condiţiile la limită, deoarece un parametru **params** poate accepta orice număr de parametri – *chiar şi nici unul*! De exemplu, ambele apeluri următoare ale lui **minVal()** sunt corecte din punct de vedere sintactic:

```
min = ob.minVal();    // fara parametri
min = ob.minVal(3);   // un parametru
```

Din acest motiv, în corpul lui **minVal()** se face o verificare, pentru a confirma că există cel puţin un element în tabloul **nums** înainte de a încerca un acces la acel element. Fără această verificare, apelul lui **minVal()** fără parametri ar fi generat o excepţie la execuţie. (Ulterior în această lucrare, când vom discuta despre excepţii, veţi învăţa o modalitate mai bună de tratare a acestui tip de erori.) Mai mult, codul metodei **minVal()** a fost scris astfel încât cazul degenerat al apelului cu un singur parametru să funcţioneze corect. În această situaţie, metoda întoarce singurul parametru primit.

Este posibil ca o metodă să aibă atât parametri obişnuiţi, cât şi un parametru de lungime variabilă. De exemplu, în programul următor, metoda **showArgs()** primeşte un parametru de tipul **string** şi un tablou **params** cu valori întregi:

```
// Utilizarea combinata a parametrilor obisnuiti cu un parametru params.

using System;

class MyClass {
  public void showArgs(string msg, params int[] nums) {
    Console.Write(msg + ": ");

    foreach(int i in nums)
      Console.Write(i + " ");

    Console.WriteLine();
  }
}

class ParamsDemo2 {
  public static void Main() {
    MyClass ob = new MyClass();

    ob.showArgs("O lista de intregi", 1, 2, 3, 4, 5);

    ob.showArgs("Inca doi intregi", 17, 20);
  }
}
```

Această metodă ia un parametru obişnuit şi unul de tip **params**.

Programul va afişa următorul text:

```
O lista de intregi: 1 2 3 4 5
Inca doi intregi: 17 20
```

În cazul în care o metodă are atât parametri obişnuiţi, cât şi un parametru **params**, acesta trebuie să fie ultimul din lista de parametri. În plus, niciodată nu poate exista mai mult de un parametru de tipul **params**.

Exerciţii la minut

- Cum puteţi crea un parametru care înlocuieşte un număr variabil de parametri efectivi?
- Poate admite o metodă mai mult de un parametru de tip **params**?
- Un parametru **params** poate fi aşezat oriunde în cadrul listei de parametri. Este adevărat sau fals?

- Pentru a crea un parametru care acoperă un număr variabil de parametri efectivi, utilizaţi modificatorul **params**.
- Nu.
- Este fals, parametrul de tip **params** trebuie să fie ultimul din listă.

Metode care întorc obiecte

O metodă poate întoarce orice tip de date, inclusiv tipuri clasă. De exemplu, în programul următor, clasa **ErrorMsg** este utilizată la raportarea erorilor. Metoda **getErrorMsg()** din această clasă întoarce un obiect **string** care conţine o descriere a erorii, în funcţie de un cod de eroare pe care îl primeşte ca parametru.

```
// Intoarcerea unui obiect string.

using System;

class ErrorMsg {
  string[] msgs = {
    "Eroare de afisare",
    "Eroare de intrare",
    "Disc plin",
    "Index in afara domeniului"
  };

  // Intoarce mesajul de eroare corespunzator.
  public string getErrorMsg(int i) {
    if(i >= 0 & i < msgs.Length)
      return msgs[i];
    else
      return "Cod de eroare incorect";
  }
}

class ErrMsg {
  public static void Main() {
    ErrorMsg err = new ErrorMsg();

    Console.WriteLine(err.getErrorMsg(2));
    Console.WriteLine(err.getErrorMsg(19));
  }
}
```

Întoarce un obiect de tipul **string**.

Acesta afişează textul de mai jos:

```
Disc plin
Cod de eroare incorect
```

Este de asemenea posibil să întoarceţi obiecte având ca tip clase create de dumneavoastră. Spre exemplu, în continuare prezentăm o versiune prelucrată a programului anterior, care creează două clase dedicate raportării erorilor. Prima se numeşte **Err** şi încapsulează un mesaj de eroare laolaltă cu un cod de severitate. Cea de-a doua se numeşte **ErrorInfo**. Aceasta defineşte o metodă numită **getErrorInfo()**, care întoarce un obiect de tipul **Err**.

```
// Intoarce un obiect definit de programator.

using System;

class Err {
  public string msg;   // mesajul de eroare
```

```
    public int severity; // cod care arata severitatea erorii

    public Err(string m, int s) {
      msg = m;
      severity = s;
    }
}

class ErrorInfo {
  string[] msgs = {
    "Eroare de afisare",
    "Eroare de intrare",
    "Disc plin",
    "Index in afara domeniului"
  };
  int[] howbad = { 3, 3, 2, 4 };

  public Err getErrorInfo(int i) { // Întoarce un obiect de tipul Err.
    if(i >= 0 & i < msgs.Length)
      return new Err(msgs[i], howbad[i]);
    else
      return new Err("Cod de eroare incorect", 0);
  }
}

class ErrInfo {
  public static void Main() {
    ErrorInfo err = new ErrorInfo();
    Err e;

    e = err.getErrorInfo(2);
    Console.WriteLine(e.msg + ", severitatea: " + e.severity);

    e = err.getErrorInfo(19);
    Console.WriteLine(e.msg + ", severitatea: " + e.severity);
  }
}
```

Iată şi rezultatul afişat:

```
Disc plin, severitatea: 2
Cod incorect de eroare, severitatea: 0
```

La fiecare apel al metodei **getErrorInfo()**, se creează un nou obiect de tip **Err**, iar o referinţă către acesta este întoarsă rutinei apelante. Acest obiect este apoi utilizat în **Main()** pentru afişarea mesajului de eroare şi codului de severitate.

Atunci când o metodă întoarce un obiect, acesta rămâne „în viaţă" până în momentul în care nu mai există referinţe către el. În acest moment, el devine obiect al colectării automate. În concluzie, obiectele nu sunt distruse doar pentru motivul că metodele care le-au creat se termină.

Supraîncărcarea metodelor

În această secţiune, veţi învăţa despre una dintre cele mai interesante facilităţi oferite de C#: supraîncărcarea metodelor. În C#, două sau mai multe metode din cadrul aceleiaşi clase pot avea acelaşi nume, atât timp cât declaraţiile parametrilor lor diferă. Când aceasta se întâmplă, metodele se numesc *supraîncărcate*, procesul purtând numele de *supraîncărcare a metodelor*. Supraîncărcarea metodelor este una dintre modalităţile prin care limbajul C# implementează conceptul de polimorfism.

În general, pentru a supraîncărca o anumită metodă, este suficient să se declare versiuni diferite ale sale. Restul este misiunea compilatorului. Trebuie însă să remarcaţi o restricţie importantă: tipul şi/sau numărul parametrilor fiecărei metode supraîncărcate trebuie să difere. Nu este suficient ca două metode să difere numai prin tipul valorii întoarse. Ele trebuie să difere şi prin tipurile sau numărul parametrilor. (Cunoscând numai tipurile rezultatelor, compilatorul de C# nu dispune întotdeauna de suficientă informaţie pentru a decide ce metodă să aleagă.) Desigur, metodele supraîncărcate pot diferi şi prin tipul rezultatului. Atunci când se apelează o metodă supraîncărcată, se alege pentru execuţie versiunea metodei ai cărei parametri se potrivesc cu parametrii efectivi.

În continuare, iată un exemplu simplu care prezintă supraîncărcarea metodelor:

```
// Demonstreaza supraincarcarea metodelor.

using System;

class Overload {
  public void ovlDemo() { // Prima versiune
    Console.WriteLine("Fara parametri");
  }

  // se supraincarca ovlDemo cu un parametru intreg.
  public void ovlDemo(int a) { // Versiunea a doua
    Console.WriteLine("Un parametru: " + a);
  }

  // supraincarcam ovlDemo cu doi parametri intregi.
  public int ovlDemo(int a, int b) { // Versiunea a treia
    Console.WriteLine("Doi parametri: " + a + " " + b);
    return a + b;
  }

  // supraincarcam ovlDemo cu doi parametri double.
  public double ovlDemo(double a, double b) { // Versiunea a patra
    Console.WriteLine("Doi parametri double: " + a + " " + b);
    return a + b;
  }
}

class OverloadDemo {
  public static void Main() {
    Overload ob = new Overload();
```

```
    int resI;
    double resD;

    // apelam toate versiunile lui ovlDemo()
    ob.ovlDemo();
    Console.WriteLine();

    ob.ovlDemo(2);
    Console.WriteLine();

    resI = ob.ovlDemo(4, 6);
    Console.WriteLine("Rezultatul lui ob.ovlDemo(4, 6): " + resI);
    Console.WriteLine();

    resD = ob.ovlDemo(1.1, 2.32);
    Console.WriteLine("Rezultatul lui ob.ovlDemo(1.1, 2.32): " + resD);
  }
}
```

Acest program va genera următoarele rezultate:

```
Fara parametri

Un parametru: 2

Doi parametri: 4 6
Rezultatul lui ob.ovlDemo(4, 6): 10

Doi parametri double: 1.1 2.32
Rezultatul lui ob.ovlDemo(1.1, 2.32): 3.42
```

După cum se observă, metoda **ovlDemo()** este definită de patru ori. Prima versiune nu acceptă nici un parametru, cea de-a doua are un parametru întreg, a treia are doi parametri întregi, iar a patra doi parametri de tipul **double**. Remarcați că primele două versiuni ale lui **ovlDemo()** întorc **void**, în timp ce ultimele două întorc câte o valoare. Aceasta este perfect corect, dar, după cum am mai explicat, supraîncărcarea nu este afectată în nici un fel de tipul rezultatului întors de o metodă. Prin urmare, dacă veți încerca să utilizați următoarele două versiuni ale metodei **ovlDemo()**, se va produce o eroare.

```
// o prima versiune ovlDemo(int) este corecta.
public void ovlDemo(int a) {
  Console.WriteLine("Un parametru: " + a);
}

/* Eroare! Prezenta a doua metode ovlDemo(int) este
   incorecta, chiar daca tipul rezultatelor intoarse difera.
*/
public int ovlDemo(int a) {
  Console.WriteLine("Un parametru: " + a);
  return a * a;
}
```

Tipul rezultatului nu poate fi utilizat pentru diferențierea metodelor supraîncărcate.

După cum arată și comentariile, diferența dintre tipurile rezultatelor întoarse este insuficientă pentru scopul supraîncărcării.

Dacă vă mai amintiți din modulul 2, limbajul C# oferă anumite conversii automate de tip. Aceste conversii se aplică de asemenea și parametrilor metodelor supraîncărcate. De exemplu, să considerăm următorul program:

```
/* Conversiile automate de tip pot afecta
   alegerea metodelor supraincarcate.
*/

using System;

class Overload2 {
  public void f(int x) {
    Console.WriteLine("Metoda f(int): " + x);
  }

  public void f(double x) {
    Console.WriteLine("Metoda f(double): " + x);
  }
}

class TypeConv {
  public static void Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;

    ob.f(i); // apeleaza ob.f(int)
    ob.f(d); // apeleaza ob.f(double)

    ob.f(b); // apeleaza ob.f(int) -- conversie de tip
    ob.f(s); // apeleaza ob.f(int) -- conversie de tip
    ob.f(f); // apeleaza ob.f(double) -- conversie de tip
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Metoda f(int): 10
Metoda f(double): 10.1
Metoda f(int): 99
Metoda f(int): 10
Metoda f(double): 11.5
```

În acest exemplu, sunt definite numai două versiuni ale metodei **f()**: una care are un parametru **int** și alta cu un parametru de tipul **double**. Este însă posibil să transmitem metodei **f()** și valori **byte**, **short** sau **float**. În cazul tipurilor **byte** și **short**, se va face automat conversia la **int**. În consecință, **f(int)** va fi apelată. În cazul tipului **float**, valoarea este convertită la **double** și se apelează **f(double)**.

Este foarte important să înţelegeţi însă că regulile de conversie automată a tipurilor se aplică numai dacă nu există o potrivire directă între parametrii formali şi cei efectivi. De exemplu, în continuare este prezentat programul anterior, la care s-a adăugat o versiune a lui `f()` care admite un parametru de tipul `byte`:

```
// Se adauga f(byte)

using System;

class Overload2 {
  public void f(byte x) {
    Console.WriteLine("Metoda f(byte): " + x);
  }

  public void f(int x) {
    Console.WriteLine("Metoda f(int): " + x);
  }

  public void f(double x) {
    Console.WriteLine("Metoda f(double): " + x);
  }
}

class TypeConv {
  public static void Main() {
    Overload2 ob = new Overload2();

    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;

    ob.f(i);  // apeleaza ob.f(int)
    ob.f(d);  // apeleaza ob.f(double)

    ob.f(b);  // apeleaza ob.f(byte) -- nu se mai face conversie

    ob.f(s);  // apeleaza ob.f(int) -- conversie de tip
    ob.f(f);  // apeleaza ob.f(double) -- conversie de tip
  }
}
```

La o nouă rulare a programului, se va afişa următorul text:

```
Metoda f(int): 10
Metoda f(double): 10.1
Metoda f(byte): 99
Metoda f(int): 10
Metoda f(double): 11.5
```

În această versiune, deoarece există o versiune a lui `f()` care primeşte un parametru de tipul `byte`, la apelul lui `f()` cu un parametru efectiv `byte`, se apelează `f(byte)` şi deci conversia automată de tip la `int` nu mai are loc.

Modificatorii `ref` şi `out` contează în alegerea uneia dintre metodele supraîncărcate. De exemplu, următoarele două secvenţe definesc două metode distincte şi separate:

```
public void f(int x) {
  Console.WriteLine("Metoda f(int): " + x);
}

public void f(ref int x) {
  Console.WriteLine("Metoda f(ref int): " + x);
}
```

Ca urmare, apelul

```
ob.f(i)
```

va invoca metoda `f(int x)`, în timp ce:

```
ob.f(ref i)
```

va invoca pe `f(ref int x)`.

Supraîncărcarea metodelor implementează conceptul de polimorfism, deoarece este una din modalităţile prin care limbajul C# implementează paradigma „o singură interfaţă, mai multe metode". Pentru a înţelege cum se realizează aceasta, să facem următorul raţionament. În limbajele care nu permit supraîncărcarea metodelor, fiecare metodă trebuie să aibă un nume unic. În mod frecvent însă, apare necesitatea implementării aceleiaşi metode pentru mai multe tipuri de date. Să ne gândim la funcţia care întoarce valoarea absolută a unui număr. În limbajele care nu permit supraîncărcarea, există de regulă trei sau chiar mai multe versiuni ale acestei funcţii, fiecare având un nume puţin diferit. De exemplu, în C, funcţia `abs()` întoarce valoarea absolută a unui întreg, `labs()` întoarce valoarea absolută a unui întreg lung, iar `fabs()` face acelaşi lucru pentru o valoare în virgulă mobilă. Deoarece limbajul C nu permite supraîncărcarea, fiecare funcţie are un nume diferit, chiar dacă toate trei fac de fapt acelaşi lucru. Aceasta complică lucrurile, din punct de vedere conceptual, mai mult decât este cazul. Deşi ideea fundamentală a fiecărei funcţii este aceeaşi, sunteţi obligat să reţineţi trei nume diferite. Această situaţie nu se mai întâlneşte şi în C#, deoarece toate metodele care calculează valoarea absolută pot partaja acelaşi nume. Într-adevăr, biblioteca standard de clase din C# conţine o metodă de calcul a valorii absolute, numită `Abs()`. Această metodă este redefinită de clasa `System.Math` din C# pentru a lucra cu toate tipurile numerice. Compilatorul de C# va determina care metodă a lui `Abs()` trebuie apelată în funcţie de tipul parametrului efectiv.

Marele beneficiu al supraîncărcării este acela că permite ca metodele înrudite să fie apelate utilizând acelaşi nume. În concluzie, numele `Abs` reprezintă *operaţia generală* care se efectuează. Rămâne la latitudinea compilatorului să aleagă versiunea *specifică* pentru fiecare caz particular. Dumneavoastră, ca programator, trebuie să reţineţi doar numele operaţiei generale. Utilizând conceptul de polimorfism, mai multe nume au fost comprimate la unul singur. Chiar dacă acesta este un exemplu banal,

dacă extindeți conceptul veți observa că supraîncărcarea vă poate ajuta să gestionați și situații mai complexe.

Sfatul expertului

Întrebare: Am auzit termenul de *signatură*, folosit de programatori în C#. Ce reprezintă acest termen?

Răspuns: În ceea ce priveşte limbajul C#, signatura reprezintă numele unei metode, urmat de lista parametrilor acesteia. În concluzie, din punctul de vedere al supraîncărcării, două metode din aceeaşi clasă nu pot avea aceeaşi signatură. Remarcați că signatura nu include şi tipul rezultatului întors, acesta nefiind utilizat în C# la rezoluția metodelor cu definiție multiplă.

Atunci când supraîncărcați o metodă, fiecare versiune a acelei metode poate efectua orice activitate doriți. Nu există nici o regulă care să oblige ca metodele supraîncărcate să fie înrudite. Din punct de vedere al stilului de programare însă, supraîncărcarea metodelor implică existența unei relații între acestea. În consecință, chiar dacă puteți utiliza acelaşi nume pentru a supraîncărca metode total diferite, nu se recomandă să o faceți. De exemplu, puteți utiliza numele **sqr** pentru a crea metode care întorc pătratul unei valori întregi și respectiv rădăcina pătrată a unei valori în virgulă mobilă. Aceste două operații sunt însă fundamental diferite. Aplicarea supraîncărcării metodelor în acest mod abuzează de scopul pe care aceasta şi-l propune. În practică, trebuie să aplicați supraîncărcarea numai pentru operații apropiate.

Exerciții la minut

- Pentru a putea supraîncărca o metodă, ce condiție trebuie îndeplinită?
- Are tipul rezultatului vreun rol la supraîncărcarea metodelor?
- Cum afectează conversia automată de tip din C# supraîncărcarea metodelor?

- Pentru ca o metodă să poată redefini o altă metodă, tipul şi/sau numărul parametrilor trebuie să difere.
- Nu. Tipul rezultatului poate fi diferit între metodele redefinite, fără însă ca să afecteze în vreun fel supraîncărcarea.
- Atunci când un set de parametri efectivi nu se potriveşte direct cu nici un set de parametri formali, se încearcă utilizarea conversiei automate de tip. Se alege versiunea cu setul cel mai apropiat de parametri formali, iar dacă parametrii efectivi pot fi convertiți automat la tipul parametrilor formali, se execută această operație.

Supraîncărcarea constructorilor

Ca şi orice alte metode, constructorii pot fi de asemenea supraîncărcați. Această facilitate vă permite să construiți obiecte într-o varietate de moduri. De exemplu, să considerăm următorul program:

```
// Prezinta un constructor supraincarcat.

using System;

class MyClass {
  public int x;

  public MyClass() {
    Console.WriteLine("Constructorul MyClass().");
    x = 0;
  }

  public MyClass(int i) {
    Console.WriteLine("Constructorul MyClass(int).");
    x = i;
  }

  public MyClass(double d) {
    Console.WriteLine("Constructorul MyClass(double).");
    x = (int) d;
  }

  public MyClass(int i, int j) {
    Console.WriteLine("Constructorul.MyClass(int, int).");
    x = i * j;
  }
}

class OverloadConsDemo {
  public static void Main() {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass(88);
    MyClass t3 = new MyClass(17.23);
    MyClass t4 = new MyClass(2, 4);

    Console.WriteLine("t1.x: " + t1.x);
    Console.WriteLine("t2.x: " + t2.x);
    Console.WriteLine("t3.x: " + t3.x);
    Console.WriteLine("t4.x: " + t4.x);
  }
}
```

Obiectele sunt construite în mai multe moduri.

Rezultatul afişat de program este prezentat mai jos:

```
Constructorul MyClass().
Constructorul MyClass(int).
Constructorul MyClass(double).
Constructorul MyClass(int, int).
t1.x: 0
t2.x: 88
```

```
t3.x: 17
t4.x: 8
```

Constructorul `MyClass()` este definit de patru ori, fiecare versiune construind un obiect într-un mod diferit. Apelul constructorului adecvat se face pe baza parametrilor specificați atunci când se execută `new`. Prin supraîncărcarea constructorului unei clase, oferiți utilizatorilor clasei flexibilitate în modul de construcție a obiectelor.

Unul dintre cele mai frecvente motive pentru supraîncărcarea constructorilor este pentru a permite unui obiect să-l inițializeze pe altul. De exemplu, să considerăm acest program care utilizează clasa `Sumare` pentru a calcula suma numerelor naturale mai mici decât un număr dat:

```
// Initializarea unui obiect cu ajutorul altuia.

using System;

class Sumare {
  public int sum;

  // Constructor cu un parametru de tipul int.
  public Sumare(int num) {
    sum = 0;
    for(int i = 1; i <= num; i++)
      sum += i;
  }

  // Constructor care utilizeaza un alt obiect.
  public Sumare(Sumare ob) {   <-- Construieşte un obiect cu ajutorul altuia.
    sum = ob.sum;
  }
}

class SumDemo {
  public static void Main() {
    Sumare s1 = new Sumare(5);
    Sumare s2 = new Sumare(s1);

    Console.WriteLine("s1.sum: " + s1.sum);
    Console.WriteLine("s2.sum: " + s2.sum);
  }
}
```

Rezultatul este prezentat mai jos:

```
s1.sum: 15
s2.sum: 15
```

Adeseori, după cum se observă și din acest exemplu, avantajul obținut punând la dispoziție un constructor care utilizează un obiect pentru inițializarea altuia este eficiența. În acest caz, atunci când se construiește `s2`, nu mai este necesar să recalculăm suma. Desigur, chiar și în cazurile în care nu se poate vorbi de o creștere a eficienței, este uneori util să punem la dispoziție un constructor care realizează o copie a unui obiect.

Apelul unui constructor redefinit utilizând `this`

Când se lucrează cu constructori supraîncărcați, este uneori util ca un constructor să îl apeleze pe altul. În C#, aceasta se realizează utilizând o altă formă a cuvântului cheie `this`. Forma generală este prezentată mai jos.

nume-constructor (*lista-parametri*): `this`(*lista-parametri-efectivi*) {
  // ... corpul constructorului, care poate fi vid
}

La execuția acestui constructor, versiunea supraîncărcată care se potrivește cu lista de parametri specificată prin *lista-parametri-efectivi* se execută mai întâi. După aceea, se execută și instrucțiunile din corpul constructorului inițial, dacă acestea există. În continuare, este prezentat un exemplu.

```
// Prezinta apelul unui constructor utilizand this.

using System;

class CoordXY {
  public int x, y;

  public CoordXY() : this(0, 0) {
    Console.WriteLine("Constructorul CoordXY()");
  }

  public CoordXY(CoordXY obj) : this(obj.x, obj.y) {
    Console.WriteLine("Constructorul CoordXY(CoordXY obj)");
  }

  public CoordXY(int i, int j) {
    Console.WriteLine("Constructorul CoordXY(int, int)");
    x = i;
    y = j;
  }
}

class OverloadConsDemo {
  public static void Main() {
    CoordXY t1 = new CoordXY();
    CoordXY t2 = new CoordXY(8, 9);
    CoordXY t3 = new CoordXY(t2);

    Console.WriteLine("t1.x, t1.y: " + t1.x + ", " + t1.y);
    Console.WriteLine("t2.x, t2.y: " + t2.x + ", " + t2.y);
    Console.WriteLine("t3.x, t3.y: " + t3.x + ", " + t3.y);
  }
}
```

Rezultatul afişat de program este prezentat mai jos:

```
Constructorul CoordXY(int, int)
Constructorul CoordXY()
Constructorul CoordXY(int, int)
Constructorul CoordXY(int, int)
Constructorul CoordXY(CoordXY obj)
```

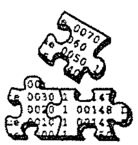
```
t1.x, t1.y: 0, 0
t2.x, t2.y: 8, 9
t3.x, t3.y: 8, 9
```

Să explicăm cum funcţionează acest program. În clasa **CoordXY**, singurul constructor care iniţializează efectiv câmpurile **x** şi **y** este **CoordXY(int, int)**. Ceilalţi doi constructori îl invocă pe **CoordXY(int, int)** prin intermediul lui **this**. De exemplu, la crearea obiectului **t1**, se apelează constructorul adecvat, **CoordXY()**. Acest apel determină executarea lui **this(0, 0)**, care în acest caz este convertit într-un apel **CoordXY(0, 0)**. Crearea lui **t2** se efectuează într-o manieră similară.

Motivul principal pentru care invocarea constructorilor supraîncărcaţi prin intermediul lui **this** se dovedeşte utilă este evitarea duplicării inutile a codului. În exemplul anterior, nu era cazul ca toţi cei trei constructori să duplice aceeaşi secvenţă de iniţializare, pe care utilizarea lui **this** o anulează. Un alt avantaj este că puteţi crea constructori cu „valori implicite", care se utilizează atunci când parametrii efectivi corespunzători sunt omişi din apel. De exemplu, un alt posibil constructor al clasei **CoordXY()** este cel prezentat mai jos:

```
public CoordXY(int x) : this(x, x) { }
```

Acest constructor iniţializează automat coordonata **y** cu valoarea coordonatei **x**. Desigur, trebuie să recurgeţi la aceste „valori implicite" cu mult discernământ, deoarece folosirea lor incorectă poate induce cu uşurinţă în eroare pe cei ce vor utiliza clasele dumneavoastră.



Proiectul 6-2: Supraîncărcarea constructorului clasei Queue

În acest proiect, veţi îmbunătăţi clasa **Queue**, adăugându-i doi noi constructori. Primul dintre aceştia va construi o nouă coadă dintr-o coadă existentă. Cel de-al doilea va construi o coadă, pe care o va şi iniţializa. După cum veţi vedea, adăugarea acestor constructori va facilita substanţial utilizarea clasei **Queue**.

Pas cu pas

1. Creaţi un fişier numit **QDemo2.cs** şi copiaţi versiunea actualizată a clasei **Queue** din proiectul 6-1 în acesta.
2. Adăugaţi următorul constructor, care creează o nouă coadă dintr-o coadă deja existentă:

```
// Construieste un obiect Queue din altul.
  public Queue(Queue ob) {
  putloc = ob.putloc;
  getloc = ob.getloc;
  q = new char[ob.q.Length];

  // se copiaza elementele din coada
  for(int i = getloc + 1; i <= putloc; i++)
    q[i] = ob.q[i];
}
```


Examinaţi cu atenţie acest constructor. Mai întâi, el iniţializează **putloc** şi **getloc** cu valorile conţinute de parametrul **ob**. Apoi, se alocă un tablou nou pentru memorarea cozii şi se copiază elementele din **ob** în acel tablou. După construire, noua coadă va fi o copie identică a celei originale, constituind însă un obiect diferit.

3. Adăugaţi constructorul care iniţializează o coadă dintr-un tablou de caractere, al cărui cod este prezentat în cele ce urmează:

```
// Construieste o coada cu valori initiale.
public Queue(char [] a) {
  putloc = 0;
  getloc = 0;
  q = new char[a.Length + 1];

  for(int i = 0; i < a.Length; i++) put(a[i]);
}
```

Acest constructor creează o coadă suficient de mare pentru a putea memora caracterele din tabloul **a** şi apoi introduce caracterele din tablou în coadă. Din cauza modului de lucru al mecanismului de coadă, lungimea cozii trebuie să depăşească lungimea tabloului cu o unitate.

4. În continuare, prezentăm codul complet al clasei **Queue** actualizate, împreună cu clasa **QDemo2**, care o utilizează:

```
// O clasa care implementeaza o coada de caractere.

using System;

class Queue {
  // acesti membri sunt acum privati
  char [] q;  // tabloul care implementeaza coada
  int putloc, getloc;  // indicii put si get

  // Construieste o coada vida de dimensiune data.
  public Queue(int size) {
    q = new char[size + 1];  // se aloca memorie pentru coada
  putloc = getloc = 0;
  }

  // Construieste un obiect Queue din altul.
  public Queue(Queue ob) {
    putloc = ob.putloc;
    getloc = ob.getloc;
    q = new char[ob.q.Length];

    // se copiaza elementele din coada
    for(int i = getloc + 1; i <= putloc; i++)
    q[i] = ob.q[i];
  }

  // Construieste o coada cu valori initiale.
  public Queue(char [] a) {
    putloc = 0;
    getloc = 0;
    q = new char[a.Length + 1];

    for(int i = 0; i < a.Length; i++) put(a[i]);
  }
```

```
    // insereaza un caracter in coada
    public void put(char ch) {
      if(putloc == q.Length - 1) {
        Console.WriteLine(" - Coada este plina.");
        return;
      }

      putloc++;
      q[putloc] = ch;
    }

    // intoarce un caracter din coada
    public char get() {
      if(getloc == putloc) {
        Console.WriteLine(" - Coada este vida.");
        return (char) 0;
      }

      getloc++;
      return q[getloc];
    }
  }

// Utilizeaza clasa Queue.
class QDemo2 {
  public static void Main() {
    // construim o coada vida cu 10 elemente
    Queue q1 = new Queue(10);

    char[] name = {'I', 'o', 'n'};
    // construim o coada dintr-un tablou
    Queue q2 = new Queue(name);

    char ch;
    int i;

    // inseram cateva caractere in q1
    for(i = 0; i < 10; i++)
      q1.put((char)('A' + i));

    // construim o coada cu ajutorul alteia
    Queue q3 = new Queue(q1);

    // Afisam cozile.
    Console.Write("Continutul lui q1: ");
    for(i = 0; i < 10; i++) {
      ch = q1.get();
      Console.Write(ch);
    }

    Console.WriteLine("\n");

    Console.Write("Continutul lui q2: ");
    for(i = 0; i < 3; i++) {
      ch = q2.get();
      Console.Write(ch);
    }
```



```
    Console.WriteLine("\n");

    Console.Write("Continutul lui q3: ");
    for(i = 0; i < 10; i++) {
      ch = q3.get();
      Console.Write(ch);
    }
  }
}
```

5. Acest program afişează rezultatul prezentat mai jos:

```
Continutul lui q1: ABCDEFGHIJ

Continutul lui q2: Ion

Continutul lui q3: ABCDEFGHIJ
```

Metoda `Main()`

Până acum, am utilizat o singură formă a metodei **`Main()`**. Există însă mai multe forme ale metodei **`Main()`**, datorate supraîncărcării. Unele dintre acestea pot întoarce o valoare, în timp ce altele pot primi parametri. Le vom studia pe rând pe fiecare.

Întoarcerea valorilor din `Main()`

La terminarea unui program, puteţi întoarce o valoare în procesul apelant (care adeseori este chiar sistemul de operare) dacă întoarceţi o valoare din **`Main()`**. Pentru aceasta, se utilizează această formă a lui **`Main()`**:

```
public static int Main()
```

Remarcaţi că în loc să fie declarată ca **`void`**, această versiune a lui **`Main()`** are un rezultat de tipul **`int`**.

De regulă, valoarea întoarsă de **`Main()`** indică dacă programul s-a terminat normal sau ca urmare a unei condiţii de eroare. Prin convenţie, o valoare întoarsă egală cu 0 indică o terminare normală. Toate celelalte valori indică apariţia unor erori.

Parametrii metodei `Main()`

Multe programe acceptă ceea ce programatorii denumesc parametri *în linia de comandă*. Un parametru în linia de comandă este o informaţie care urmează imediat după numele programului în linia de comandă utilizată la execuţia acestuia. Programele C# transmit aceşti parametri metodei **`Main()`**. Pentru a primi aceşti parametri, trebuie să utilizaţi una dintre următoarele forme ale lui **`Main()`**:

```
public static void Main(string[] args)

public static int Main(string[] args)
```

Prima dintre aceste forme întoarce **void**; cea de-a doua se poate utiliza pentru a întoarce o valoare întreagă. Ambele forme memorează parametrii din linia de comandă ca elemente în tabloul de tip **string** transmis metodei **Main()**.

Spre exemplu, programul următor afişează toţi parametrii din linia de comandă cu care este apelat:

```
// Afiseaza toata informatia din linia de comanda.
using System;

class CLDemo {
  public static void Main(string[] args) {
    Console.WriteLine("Sunt " + args.Length + " parametri.");

    Console.WriteLine("Acestia sunt: ");
    for(int i = 0; i < args.Length; i++)
      Console.WriteLine(args[i]);
  }
}
```

La execuţia programului **CLDemo** cu linia de comandă:

```
CLDemo unu doi trei
```

se va afişa următorul text:

```
Sunt 3 parametri.
Acestia sunt:
unu
doi
trei
```

Pentru a vă face o idee despre modul în care se pot utiliza practic parametrii din linia de comandă, să considerăm programul următor. Acesta primeşte un parametru în linia de comandă care precizează numele unei persoane. Programul caută apoi numele într-un tablou bidimensional de stringuri. Dacă găseşte o potrivire, programul afişează numărul de telefon al persoanei respective.

```
// O agenda telefonica automata.
using System;

class Phone {
  public static int Main(string[] args) {
    string[,] numbers = {
      { "Ion", "555-3322" },
      { "Maria", "555-8976" },
      { "Gheorghe", "555-1037" },
      { "Elena", "555-1400" },
      { "", "" },    // tabloul se termina cu siruri vide
    };
    int i;

    if(args.Length != 1) {
      Console.WriteLine("Utilizare: Phone <nume>");
      return 1;   // indicam terminarea programului prin utilizarea improprie
    }
```



```
    else {
      for(i = 0; numbers[i, 0] != ""; i++) {
        if(numbers[i, 0] == args[0]) {
          Console.WriteLine(numbers[i, 0] + ": " + numbers[i, 1]);
          break;
        }
      }
      if(numbers[i, 0] == "")
        Console.WriteLine("Numele nu exista in agenda.");
    }
    return 0;
  }
}
```

Iată şi un exemplu de rulare a programului:

```
C>Phone Maria
Maria: 555-8976
```

Există două aspecte interesante în acest program. Mai întâi, remarcaţi modul în care programul verifică prezenţa parametrului în linia de comandă înainte de a continua execuţia. Această facilitate este foarte importantă şi se poate generaliza. Dacă un program se bazează pe existenţa unuia sau mai multor parametri în linia de comandă, trebuie întotdeauna să se asigure că aceştia au fost introduşi şi au valori adecvate. Eşecul acestei operaţii conduce adeseori la terminarea anormală a programului.

În al doilea rând, remarcaţi că programul întoarce un cod de terminare. Dacă parametrul din linia de comandă necesar nu este prezent, programul întoarce 1, indicând o terminare anormală. În caz contrar, programul întoarce 0 la terminare.

Exerciţii la minut

- Poate un constructor avea ca parametru un obiect din clasa din care face parte?
- De ce se recomandă să puneţi la dispoziţie constructori supraîncărcaţi?
- Prezentaţi forma metodei **Main()** care poate primi parametri în linia de comandă, dar care nu întoarce valori.

- Da.
- Constructorii supraîncărcaţi oferă comoditate şi flexibilitate utilizatorului clasei dumneavoastră.
- Forma metodei **Main()** care poate accepta parametri în linia de comandă, dar nu întoarce valori, este:
  `public static void Main(string[] args)`

Recursivitate

În C#, o metodă se poate autoapela. Acest proces este denumit *recursivitate*, iar metodele care se autoapelează se numesc *recursive*. În general, recursivitatea este operația prin care un concept este definit în funcție de sine însuși, fiind oarecum similară cu o definiție circulară. Componenta esențială a oricărei metode recursive este o instrucțiune care execută un apel către metoda în care este conținută. Recursivitatea reprezintă un mecanism de control puternic.

Exemplul cel mai uzual de recursivitate este cel al calculului factorialului unui număr întreg. Factorialul unui număr întreg N este produsul tuturor numerelor întregi cuprinse între 1 și N. De exemplu, factorialul lui 3 este egal cu 1×2×3, adică 6. Programul următor prezintă o modalitate recursivă de calcul al factorialului unui număr întreg. Pentru a avea un termen de comparație, este prezentată și o metodă nerecursivă echivalentă.

```
// Un exemplu simplu de recursivitate.

using System;

class Factorial {
  // Aceasta este o functie recursiva.
  public int factR(int n) {
    int result;

    if(n == 1) return 1;
    result = factR(n - 1) * n;   <-- Apel recursiv al metodei factR().
    return result;
  }

  // O versiune iterativa echivalenta.
  public int factI(int n) {
    int t, result;

    result = 1;
    for(t = 1; t <= n; t++) result *= t;
    return result;
  }
}

class Recursivitate {
  public static void Main() {
    Factorial f = new Factorial();

    Console.WriteLine("Factoriale calculate cu metoda recursiva.");
    Console.WriteLine("Factorialul lui 3 este " + f.factR(3));
    Console.WriteLine("Factorialul lui 4 este " + f.factR(4));
    Console.WriteLine("Factorialul lui 5 este " + f.factR(5));
    Console.WriteLine();

    Console.WriteLine("Factoriale calculate cu metoda iterativa.");
    Console.WriteLine("Factorialul lui 3 este " + f.factI(3));
    Console.WriteLine("Factorialul lui 4 este " + f.factI(4));
```



```
    Console.WriteLine("Factorialul lui 5 este " + f.factI(5));
  }
}
```

Rezultatul afișat de acest program este prezentat mai jos:

```
Factoriale calculate cu metoda recursiva.
Factorialul lui 3 este 6
Factorialul lui 4 este 24
Factorialul lui 5 este 120

Factoriale calculate cu metoda iterativa.
Factorialul lui 3 este 6
Factorialul lui 4 este 24
Factorialul lui 5 este 120
```

Modul de operare al metodei **factI()** este evident. Aceasta utilizează o buclă care pornește de la 1 și înmulțește pe rând fiecare număr cu produsul calculat anterior.

Funcționarea metodei recursive **factR()** este ceva mai complicată. Când este apelată cu parametrul 1, metoda **factR()** întoarce 1; în caz contrar, va întoarce produsul **factR(n - 1) * n**. Pentru evaluarea acestei expresii, se apelează **factR()** cu parametrul **n - 1**. Operația se repetă până când **n** devine 1 și apelurile metodei încep să revină. De exemplu, la calculul factorialului lui 2, primul apel al lui **factR()** va genera un al doilea apel, în care parametrul metodei este 1. Acest apel va întoarce 1, care se înmulțește cu 2 (valoarea inițială a lui **n**). Rezultatul final este deci 2. Este interesant să introduceți instrucțiuni **WriteLine()** în corpul metodei **factR()** pentru a afișa nivelul fiecărui apel, precum și rezultatul intermediar.

Atunci când o metodă se autoapelează, pentru noile variabile locale și noii parametri se alocă spațiu pe stiva compilatorului, iar corpul metodei se execută de la început cu aceste valori. Un apel recursiv nu face o nouă copie a metodei. Doar parametrii sunt cei care se modifică. La revenirea unui apel recursiv, vechile valori ale variabilelor locale și parametrilor se scot din stivă, iar execuția continuă de la punctul apelului recursiv. Metodele recursive simulează astfel funcția unui telescop, care apropie și apoi depărtează o imagine.

Versiunile recursive ale multor rutine se execută ceva mai lent decât echivalentele lor iterative din cauza întârzierilor adăugate de apelurile suplimentare de metode. Un număr prea mare de apeluri recursive ale unei metode poate determina o depășire a stivei. Deoarece spațiul pentru parametrii și variabilele locale este rezervat pe stivă, iar fiecare nou apel creează încă o copie a acestor variabile, se poate întâmpla ca stiva să fie epuizată. În acest caz, motorul de programare din C# va genera o excepție. De regulă însă, aceasta nu se întâmplă decât în cazurile nefericite ale unor rutine recursive scăpate de sub control.

Principalul avantaj al recursivității este că anumite tipuri de algoritmi se implementează mult mai clar și mai simplu recursiv decât iterativ. De exemplu, algoritmul de sortare rapidă este destul de dificil de implementat în mod iterativ. De asemenea,

unele categorii de probleme, îndeosebi cele de inteligență artificială, par a fi dedicate soluțiilor recursive.

Orice metodă recursivă pe care o scrieți trebuie să conțină o instrucțiune condițională (denumită *punct fix*), ca de exemplu un **if**, care să forțeze revenirea metodei fără executarea unui apel recursiv. Dacă omiteți această condiție, orice apel al metodei nu va reveni niciodată. Acest tip de erori este foarte frecvent când se lucrează cu recursivitatea. Utilizați cu generozitate instrucțiunile **WriteLine()** pentru a putea urmări ce se întâmplă, oprind execuția programului dacă observați că ați greșit.

Cuvântul cheie static

Există situații în care aveți nevoie să definiți un membru al unei clase care să fie utilizat fără a depinde de vreo instanță a clasei. În mod obișnuit, un membru al unei clase trebuie să fie accesat utilizând o instanță a clasei, dar este posibil să creăm un membru care să fie utilizat de sine stătător, fără a referi o anumită instanță. Pentru a crea un astfel de membru, declarația acestuia trebuie precedată de cuvântul cheie **static**. Dacă un membru este declarat ca **static**, poate fi accesat înaintea creării oricărei instanțe a clasei și fără a referi vreun obiect. Atât metodele, cât și variabilele pot fi declarate **static**. Cel mai uzual exemplu de membru **static** este metoda **Main()**, care este declarată astfel deoarece trebuie să fie apelată de sistemul de operare la începutul programului.

În afara clasei, pentru a utiliza un membru **static**, trebuie să precizați numele clasei, urmat de operatorul punct. Nu este necesar să creați nici o instanță. De fapt, membrii **static** nu pot fi accesați prin intermediul instanțelor. Aceștia trebuie să fie precedați de numele clasei. Spre exemplu, pentru a atribui valoarea 10 variabilei **static** numită **count** din clasa **Timer**, utilizați linia următoare:

```
Timer.count = 10;
```

Acest format seamănă cu cel utilizat la accesul variabilelor instanță obișnuite prin intermediul unui obiect, înlocuind însă numele instanței cu numele clasei. O metodă **static** poate fi apelată în același mod – utilizând operatorul punct după numele clasei.

Variabilele declarate ca **static** sunt, de fapt, variabile globale. La declararea unei instanțe a clasei, variabilele **static** nu sunt copiate. În mod contrar, variabilele **static** sunt partajate de toate instanțele clasei. Variabilele **static** sunt inițializate la încărcarea clasei în memorie. Dacă nu se specifică în mod explicit o valoare de inițializare, o variabilă **static** se inițializează cu zero dacă este de un tip numeric, cu null în cazul tipurilor referință, respectiv cu **false** pentru tipul **bool**. O variabilă **static** are deci întotdeauna o valoare.

Diferența dintre o metodă **static** și o metodă obișnuită este că metoda **static** poate fi apelată fiind prefixată cu numele clasei din care face parte, fără crearea unei

instanțe a clasei. Ați văzut deja un exemplu de astfel de metodă: metoda **Sqrt()**, care este declarată **static** în clasa **System.Math** din C#.

În continuare, este prezentat un exemplu care creează o variabilă **static** și o metodă **static**:

```
// Utilizarea lui static.

using System;

class StaticDemo {
  // o variabila statica
  public static int val = 100;

  // o metoda statica
  public static int valDiv2() {
    return val/2;
  }
}

class SDemo {
  public static void Main() {
    Console.WriteLine("Valoarea initiala a lui StaticDemo.val
                      este " + StaticDemo.val);

    StaticDemo.val = 8;
    Console.WriteLine("StaticDemo.val este acum " + StaticDemo.val);
    Console.WriteLine("StaticDemo.valDiv2(): " + StaticDemo.valDiv2());
  }
}
```

Rezultatul afișat este prezentat mai jos:

```
Valoarea initiala a lui StaticDemo.val este 100
StaticDemo.val este acum 8
StaticDemo.valDiv2(): 4
```

După cum arată și acest rezultat, variabilele **static** se inițializează la începutul programului, înaintea creării oricărei instanțe a clasei.

Există câteva restricții care se aplică pentru metodele **static**:

- Metodele **static** nu dispun de referința **this**.
- Metodele **static** pot apela în mod direct numai alte metode **static**. Ele nu pot apela direct metodele obișnuite din aceeași clasă. Motivul este că metodele obișnuite lucrează asupra instanțelor particulare ale clasei, care nu sunt accesibile metodelor **static**.
- Metodele **static** pot accesa în mod direct numai variabile **static**. Ele nu pot utiliza variabile instanță, deoarece nu au acces la instanțele clasei.

De exemplu, în clasa următoare, metoda statică **valImpNum()** este incorectă:

```
class StaticError {
  int num = 3; // variabila instanta obisnuita
  static int val = 1024; // variabila statica
```

```
  /* Eroare! Nu putem accesa direct o variabila ne-statica dintr-o metoda
     statica. */
  static int valImpNum() {
    return val/num; // nu se va compila !
  }
}
```

Mai sus, **num** este o variabilă instanță obișnuită și nu este deci accesibilă dintr-o metodă **static**. Variabila **val** este însă corect utilizată, fiind declarată ca **static**.

Aceeași problemă apare atunci când apelăm o metodă ne-statică dintr-o metodă **static** din aceeași clasă. De exemplu:

```
using System;

class AnotherStaticError {
  // metoda ne-statica.
  void nonStaticMeth() {
    Console.WriteLine("Metoda nonStaticMeth().");
  }

  /* Eroare! Nu putem apela direct o metoda ne-statica
     dintr-o metoda statica. */
  static void staticMeth() {
    nonStaticMeth(); // nu se va compila !
  }
}
```

În acest caz, încercarea de a apela o metodă ne-statică (adică o metodă instanță) dintr-o metodă **static** este cea care va determina producerea unei erori la compilare.

Este foarte important să înțelegeți că o metodă **static** *poate* apela metodele instanță și poate accesa variabilele instanță din aceeași clasă, dar numai prin intermediul unui obiect din acea clasă. Fără a fi precedate de un obiect, acestea nu sunt accesibile. De exemplu, secvența următoare este perfect corectă:

```
class MyClass {
  // metoda ne-statica.
  void nonStaticMeth() {
    Console.WriteLine("Metoda nonStaticMeth().");
  }

  /* Putem apela o metoda ne-statica dintr-o metoda statica
     utilizand o referinta catre un obiect. */
  static void staticMeth(MyClass ob) {
    ob.nonStaticMeth(); // este corect
  }
}
```


Exerciții la minut

- Definiți recursivitatea.
- Explicați diferența dintre variabilele **static** și variabilele instanță.
- Poate o metodă **static** să apeleze o metodă ne-statică din aceeași clasă fără să utilizeze o referință către un obiect?

Proiectul 6-3: Sortarea rapidă

În modulul 5, v-am prezentat o metodă simplă de sortare, denumită metoda bulelor. Am menționat acolo că există metode sensibil mai performante. În continuare, veți dezvolta o versiune a uneia dintre cele mai bune metode: sortarea rapidă (quicksort). Sortarea rapidă, descoperită și denumită astfel de C.A.R. Hoare, este cel mai bun algoritm de sortare de uz general disponibil în momentul de față. Motivul pentru care nu l-am putut prezenta în modulul 5 este că implementările cele mai bune ale sortării rapide folosesc recursivitatea. Versiunea pe care o vom dezvolta în continuare sortează un tablou de caractere, dar poate fi adaptată pentru a sorta orice tip de obiecte doriți.

Sortarea rapidă se bazează pe ideea de *partiționare*. Procedeul general este de a selecta o valoare, denumită *pivot*, după care tabloul se partiționează în două secțiuni. Toate elementele mai mari sau egale cu pivotul sunt mutate într-o partiție, în timp ce valorile mai mici sunt mutate în cealaltă partiție. Acest procedeu se repetă în mod recursiv pentru fiecare secțiune în parte, până la sortarea completă a tabloului. De exemplu, considerând tabloul **fedacb** și luând **d** ca valoare pivot, primul pas al sortării rapide va rearanja tabloul după cum urmează:

```
Initial    f e d a c b
Pasul 1    b c a d e f
```

Procedeul se repetă apoi în fiecare secțiune, adică **bca** și **def**. După cum puteți observa, metoda este prin natura ei recursivă și, în mod evident, cea mai clară implementare a sortării rapide va fi printr-o metodă recursivă.

Selecția pivotului se poate realiza în două moduri. Acesta poate fi ales la întâmplare, sau ca medie a unui set restrâns de valori din tablou. Pentru o sortare optimală, trebuie selectat elementul care se află exact la mijlocul domeniului de valori. Determinarea acestuia nu este însă foarte simplă pentru majoritatea seturilor de date de sortat. În cazul cel mai defavorabil, valoarea aleasă se găsește într-una din extremitățile domeniului. Chiar și așa, sortarea rapidă continuă să funcționeze corect. Versiunea de sortare rapidă pe care o vom dezvolta selectează ca pivot elementul aflat la mijlocul tabloului.

Pas cu pas

1. Creați un fișier numit **QSDemo.cs**.
2. Creați clasa **Quicksort** prezentată mai jos:

```
// O versiune simpla a sortarii rapide.

using System;
```

- Recursivitatea este procesul prin care o metodă se autoapelează.
- Fiecare instanță a clasei deține o copie proprie a variabilelor instanță definite în cadrul clasei. Variabilele **static** sunt partajate de toate instanțele clasei.
- Nu.

```
class Quicksort {

  // un apel initial al metodei care face efectiv sortarea.
  public static void qsort(char[] items) {
    qs(items, 0, items.Length - 1);
  }

  // O versiune recursiva a sortarii rapide pentru caractere.
  static void qs(char[] items, int left, int right) {
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[(left + right) / 2];
    do {
      while((items[i] < x) && (i < right)) i++;
      while((x < items[j]) && (j > left)) j--;

      if(i <= j) {
        y = items[i];
        items[i] = items[j];
        items[j] = y;
        i++; j--;
      }
    } while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
  }
}
```

Pentru simplitatea interfeței programului de sortare rapidă, clasa **Quicksort** pune la dispoziție metoda **qsort()**, care execută un apel inițial al metodei care implementează de fapt operația de sortare, **qs()**. Aceasta permite ca sortarea rapidă să fie apelată doar cu numele tabloului de sortat, fără a mai specifica o partiție inițială. Deoarece metoda **qs()** este utilizată numai intern, este în mod implicit privată.

3. Pentru utilizarea clasei **Quicksort**, este suficient să apelați **Quicksort.qsort()**. Deoarece metoda **qsort()** este statică, trebuie precedată de numele clasei la apel și nu de numele unei instanțe. Nu este deci nevoie să creați un obiect de tipul **Quicksort**. După revenirea apelului, tabloul va fi sortat. Rețineți că versiunea de față funcționează numai pentru tablouri de caractere, dar puteți adapta cu ușurință ideea pentru a sorta orice tip de obiecte doriți.
4. Mai jos, prezentăm un program care utilizează clasa **Quicksort**.

```
// O versiune simpla a sortarii rapide.

using System;

class Quicksort {

  // un apel initial al metodei care face efectiv sortarea.
  public static void qsort(char[] items) {
    qs(items, 0, items.Length - 1);
  }

  // O versiune recursiva a sortarii rapide pentru caractere.
```



```
  static void qs(char[] items, int left, int right) {
    int i, j;
    char x, y;

    i = left; j = right;
    x = items[(left + right) / 2];
    do {
      while((items[i] < x) && (i < right)) i++;
      while((x < items[j]) && (j > left)) j--;

      if(i <= j) {
        y = items[i];
        items[i] = items[j];
        items[j] = y;
        i++; j--;
      }
    } while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
  }
}

class QSDemo {
  public static void Main() {
    char[] a = { 'd', 'x', 'a', 'r', 'p', 'j', 'i' };
    int i;

    Console.Write("Tabloul initial: ");
    for(int i = 0; i < a.Length; i++)
      Console.Write(a[i]);

    Console.WriteLine();

    // sortam tabloul
    Quicksort.qsort(a);

    Console.Write("Tabloul sortat: ");
    for(int i = 0; i < a.Length; i++)
      Console.Write(a[i]);
  }
}
```

Verificarea cunoștințelor

1. Dându-se fragmentul:

```
class X {
  int count;
```

este corectă secvența următoare?

```
class Y {
  public static void Main() {
    X ob = new X();

    ob.count = 10;
```

2. Un specificator de acces trebuie să ______________ declarația unui membru.
3. O structură complementară cozii este structura de stivă. Aceasta folosește disciplina de acces „primul sosit, ultimul servit", fiind de multe ori similară cu un teanc de farfurii. Prima farfurie pusă pe masă este ultima care va fi utilizată. Creați o clasă numită **Stack** care implementează o stivă care memorează caractere. Denumiți metodele de acces la stivă **push()** și **pop()**. Permiteți utilizatorului să specifice dimensiunea stivei la creare. Ceilalți membri ai clasei **Stack** trebuie să rămână privați. Indicație: Puteți utiliza clasa **Queue** ca model; trebuie doar să schimbați maniera de acces la date.
4. Se dă clasa:

```
class Test {
  int a;
  Test(int i) { a = i; }
}
```

   Să se scrie o metodă numită **swap()** care interschimbă conținutul a două obiecte de tipul **Test**.
5. Este următorul fragment corect?

```
class X {
  int meth(int a, int b) { … }
  string meth(int a, int b) { … }
```

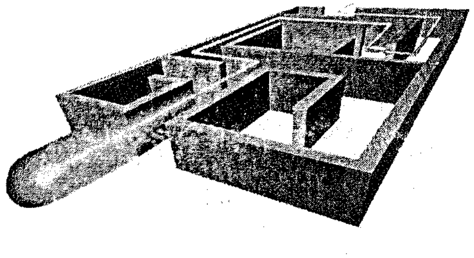
6. Scrieți o metodă recursivă care afișează conținutul unui **string** în ordine inversă.
7. Dacă o variabilă trebuie să fie partajată de toate instanțele unei clase, cum trebuie declarată?
8. Ce rol au **ref** și **out**? Ce diferență există între ei?
9. Prezentați cele patru forme ale metodei **Main()**.
10. Dându-se fragmentul de mai jos, care din următoarele apeluri sunt corecte?

```
void meth(int i, int j, params int[] args) { // …
```

   A. `meth(10, 12, 19);`
   B. `meth(10, 12, 19, 100);`
   C. `meth(10, 12, 19, 100, 200);`
   D. `meth(10, 12);`

Modulul 7

Supraîncărcarea operatorilor, indexări și proprietăți



Scopurile acestui modul

- **Noțiuni de bază despre supraîncărcarea operatorilor**
- **Supraîncărcarea operatorilor binari**
- **Supraîncărcarea operatorilor unari**
- **Supraîncărcarea operatorilor relaționali**
- **Utilizarea indexărilor**
- **Crearea proprietăților**

Acest modul studiază trei categorii speciale de membri ai unei clase: operatori supraîncărcați, indexări și proprietăți. Fiecare dintre aceste entități mărește puterea unei clase, îi facilitează utilizarea și integrarea în ierarhia de tipuri din C# și îi sporește flexibilitatea. Utilizând acești membri, este posibil să creați clase care arată și se comportă similar cu tipurile predefinite. Această *extensibilitate a tipurilor* constituie o parte însemnată a puterii unui limbaj orientat spre obiecte, cum este și C#.

Supraîncărcarea operatorilor

Limbajul C# vă permite să redefiniți semnificația unui operator în contextul dat de o anumită clasă pe care ați creat-o. Acest proces poartă numele de *supraîncărcare a operatorilor*. Supraîncărcând un operator, îi extindeți aria de utilizare la clasa dumneavoastră. Efectele operatorului sunt complet sub controlul dumneavoastră și pot diferi de la o clasă la alta. De exemplu, o clasă care definește o listă înlănțuită poate utiliza operatorul + pentru adăugarea unui obiect în listă. O clasă care implementează o stivă poate utiliza același operator + pentru adăugarea unui obiect în stivă. Alte clase pot utiliza operatorul + într-un mod cu totul diferit.

La supraîncărcarea unui operator, semnificația inițială a acestuia se conservă în întregime. Procesul de supraîncărcare reprezintă de fapt adăugarea unei noi operații, specifice unei anumite clase. În consecință, prin supraîncărcarea lui + pentru adăugarea elementelor într-o listă, de exemplu, semnificația acestuia în contextul numerelor întregi (care este de adunare) nu se modifică.

Unul din avantajele principale ale supraîncărcării operatorilor este că vă permite să integrați perfect o nouă clasă în mediul dumneavoastră de programare. După ce ați definit semnificația operatorilor în contextul unei anumite clase, puteți lucra cu obiectele acelei clase utilizând sintaxa expresiilor obișnuite din C#. Puteți chiar să utilizați un obiect în expresii care implică și alte tipuri de date.

Supraîncărcarea operatorilor este strâns legată de supraîncărcarea metodelor. Pentru a supraîncărca un operator, veți utiliza cuvântul cheie **operator** pentru a defini o *metodă operator.* Metodele operator precizează acțiunile efectuate de operatori.

Formele generale ale metodelor operator

Există două forme distincte de metode operator: una corespunzătoare operatorilor unari și cealaltă pentru operatorii binari. Formele generale pentru fiecare sunt prezentate mai jos:

```
// Forma generala pentru supraincarcarea operatorilor unari.
public static tip-rez operator op(tip-param operand)
{
    // operatii
}

// Forma generala pentru supraincarcarea operatorilor binari.
public static tip-rez operator op(tip-param1 operand1,
tip-param2 operand2)
{
    // operatii
}
```

Operatorul supraîncărcat, ca de exemplu + sau /, va substitui *op* în relațiile de mai sus. Tipul valorii întoarse de operația specificată este precizat de *tip-rez*. Chiar dacă acesta poate fi orice tip, valoarea întoarsă este adeseori de același tip cu clasa pentru care are loc supraîncărcarea operatorului. Această corespondență facilitează utilizarea operatorului supraîncărcat în cadrul expresiilor. Pentru operatorii unari, unicul operand este specificat de *operand*. În cazul operatorilor binari, operanzii sunt specificați prin *operand1* și *operand2*.

Pentru operatorii unari, operandul trebuie să fie de același tip cu clasa pentru care se redefinește operatorul. În cazul operatorilor binari, cel puțin unul dintre operanzi trebuie să fie de același tip cu clasa operatorului. În consecință, nu veți putea supraîncărca nici un operator C# pentru obiecte pe care nu le-ați creat dumneavoastră. De exemplu, nu puteți redefini operatorul + pentru **int** sau **string**.

Un alt aspect care trebuie reținut este că parametrii operatorilor nu pot utiliza modificatorii **ref** și **out**.

Supraîncărcarea operatorilor binari

Pentru a vedea cum funcționează supraîncărcarea operatorilor, vom începe cu un exemplu care redefinește doi operatori binari, + și -. Programul următor creează o clasă numită **TreiD**, care gestionează coordonatele unui obiect într-un spațiu tridimensional. Operatorul + redefinit adaugă coordonatele individuale ale unui obiect **TreiD** la coordonatele altui obiect. Operatorul – scade coordonatele unui obiect din coordonatele altui obiect.

```
// Un exemplu de supraincarcare a operatorilor.

using System;

// O clasa cu coordonate 3D
class TreiD {
  int x, y, z; // coordonate 3D

  public TreiD() { x = y = z = 0; }
  public TreiD(int i, int j, int k) { x = i; y = j; z = k; }

  // redefinirea operatorului binar +.
  public static TreiD operator +(TreiD op1, TreiD op2) {
    TreiD result = new TreiD();

    /* Se aduna coordonatele celor doua puncte, intorcand result. */
    result.x = op1.x + op2.x;  // acestea sunt adunari intregi
    result.y = op1.y + op2.y;  // iar + isi pastreaza sensul
    result.z = op1.z + op2.z;  // original relativ la ele.

    return result;
  }

  // redefinirea operatorului binar -.
  public static TreiD operator -(TreiD op1, TreiD op2)
  {
    TreiD result = new TreiD();

    /* Remarcati ordinea operanzilor. op1 este operandul stang,
       iar op2 este cel drept. */
    result.x = op1.x - op2.x;  // acestea sunt scaderi intregi
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;

    return result;
  }

  // Afiseaza coordonatele x, y, z.
  public void show() {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TreiDDemo {
  public static void Main() {
    TreiD a = new TreiD(1, 2, 3);
```

Operatorul + redefinit pentru obiecte **TreiD**.

Operatorul - redefinit pentru obiecte **TreiD**.

```
        TreiD b = new TreiD(10, 10, 10);
        TreiD c = new TreiD();
        Console.Write("Punctul a: ");
        a.show();
        Console.WriteLine();

        Console.Write("Punctul b: ");
        b.show();
        Console.WriteLine();

        c = a + b; // se aduna a si b
        Console.Write("Rezultatul a + b: ");
        c.show();
        Console.WriteLine();

        c = a + b + c; // se aduna a, b si c
        Console.Write("Rezultatul a + b + c: ");
        c.show();
        Console.WriteLine();

        c = c - a; // se scade a
        Console.Write("Rezultatul c - a: ");
        c.show();
        Console.WriteLine();

        c = c - b; // se scade b
        Console.Write("Rezultatul c - b: ");
        c.show();
        Console.WriteLine();
    }
}
```

Utilizarea obiectelor **TreiD** în expresii cu operatorii + și -

Acest program afișează următoarele rezultate:

```
Punctul a: 1, 2, 3

Punctul b: 10, 10, 10

Rezultatul a + b: 11, 12, 13

Rezultatul a + b + c: 22, 24, 26

Rezultatul c - a: 21, 22, 23

Rezultatul c - b: 11, 12, 13
```

Să studiem programul anterior cu atenție, începând cu operatorul supraîncărcat +. Atunci când două obiecte de tipul **TreiD** sunt prelucrate de operatorul +, valorile coordonatelor corespunzătoare se adună, după cum se observă și din corpul metodei **operator +()**. Remarcați însă că această metodă nu modifică valoarea niciunuia dintre operanzi. Pentru aceasta, metoda întoarce un nou obiect de tipul **TreiD**, care conține rezultatul operației. Pentru a înțelege de ce operația + nu modifică conținutul celor două obiecte, faceți o analogie cu operația aritmetică standard +, ca în exemplul 10 + 12. Rezultatul acestei operații este 22, dar atât 10, cât și 12, rămân

nemodificate. Chiar dacă nu există o regulă strictă care să împiedice operatorii supraîncărcați să modifice valoarea unuia dintre operanzi, se recomandă ca acțiunile efectuate de operatorii redefiniți să nu se abată de la semnificația lor uzuală.

După cum am precizat, metoda **operator+()** întoarce un obiect de tipul **TreiD**. Metoda putea întoarce orice alt tip valid din C#, dar faptul că întoarce un obiect **TreiD** permite operatorului + să fie utilizat în expresii compuse, de genul **a+b+c**. În acest caz, operația **a+b** generează un rezultat care este de tipul **TreiD**. Această valoare poate fi ulterior adunată cu **c**. Indiferent ce alt tip ar fi întors expresia **a+b**, adunarea ulterioară cu **c** nu ar fi funcționat.

Iată un alt aspect important: în corpul metodei **operator+()**, adunările coordonatelor individuale sunt adunări de numere întregi. Aceasta se întâmplă deoarece coordonatele individuale **x**, **y** și **z** sunt cantități întregi. Redefinirea operatorului + pentru elemente de tipul **TreiD** nu afectează operația + asupra valorilor întregi.

Să examinăm acum metoda **operator-()**. Operatorul – funcționează la fel ca și +, exceptând faptul că ordinea parametrilor contează. Amintiți-vă că adunarea este o operație comutativă, în timp ce scăderea nu este. (Adică, A – B nu este același lucru cu B – A!) Pentru toți operatorii binari, primul parametru al metodei operator conține operandul stâng. Operandul drept este conținut de către cel de-al doilea parametru. La implementarea versiunilor supraîncărcate ale operatorilor necomutativi, trebuie să vă amintiți exact care este operandul stâng și care este cel drept.

Supraîncărcarea operatorilor unari

Operatorii unari sunt supraîncărcați la fel ca și cei binari. Diferența esențială, desigur, este aceea că în acest caz avem un singur operand. De exemplu, în continuare prezentăm o metodă care redefinește operatorul unar minus în clasa **TreiD**:

```
// Supraincarcarea minusului unar.
public static TreiD operator -(TreiD op)
{
  TreiD result = new TreiD();

  result.x = -op.x;
  result.y = -op.y;
  result.z = -op.z;

  return result;
}
```

Mai sus, se creează un nou obiect care conține câmpurile operandului cu semnul schimbat. Metoda întoarce apoi acest obiect. Remarcați că operandul rămâne nemodificat. Și aici, aceasta este în concordanță cu semnificația operatorului unar minus. De exemplu, într-o expresie de forma:

```
a = -b
```

a va deveni opusul lui **b**, dar fără ca **b** să fie modificat.

Există însă două situații în care metoda operator trebuie să modifice conținutul unui operand: ++ *și* - -. Deoarece acești operatori semnifică în mod uzual incrementarea, respectiv decrementarea, variantele supraîncărcate ale lui ++ și - - trebuie să incrementeze, respectiv să decrementeze valoarea operandului. Așadar, la supraîncărcarea acestor doi operatori, valoarea operandului va fi de regulă modificată. De exemplu, iată o versiune a metodei `operator++()` pentru clasa `TreiD`:

```
// Supraincarcarea operatorului unar ++.
  public static TreiD operator ++(TreiD op)
{
  // la incrementare, modificam parametrul
  op.x++;
  op.y++;
  op.z++;

  return op;  // intoarcem operandul modificat
}
```

Operatorul ++ își modifică operandul.

Mai jos, prezentăm o versiune extinsă a programului anterior, care utilizează în plus minusul unar și operatorul de incrementare ++:

```
// Mai multi operatori redefiniti.

using System;

// O clasa cu coordonate 3D
class TreiD {
  int x, y, z;      // coordonate 3D

  public TreiD() { x = y = z = 0; }
  public TreiD(int i, int j, int k) { x = i; y = j; z = k; }

  // redefinirea operatorului binar +.
  public static TreiD operator +(TreiD op1, TreiD op2)
  {
    TreiD result = new TreiD();

    /* Se aduna coordonatele celor doua puncte, intorcand result. */
    result.x = op1.x + op2.x;  // acestea sunt adunari intregi
    result.y = op1.y + op2.y;  // iar + isi pastreaza sensul
    result.z = op1.z + op2.z;  // original relativ la ele.

    return result;
  }

  // redefinirea operatorului binar -.
  public static TreiD operator -(TreiD op1, TreiD op2) {
    TreiD result = new TreiD();

    /* Remarcati ordinea operanzilor. op1 este operandul stang,
       iar op2 este cel drept. */
    result.x = op1.x - op2.x;  // acestea sunt scaderi intregi
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;

    return result;
```



```
  }

  // Supraincarcarea minusului unar.
  public static TreiD operator -(TreiD op) {
    TreiD result = new TreiD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
  }

  // Supraincarcarea operatorului unar ++.
  public static TreiD operator ++(TreiD op)
  {
    // la incrementare, modificam parametrul
    op.x++;
    op.y++;
    op.z++;

    return op;  // intoarcem operandul modificat
  }

  // Afiseaza coordonatele x, y, z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TreiDDemo {
  public static void Main() {
    TreiD a = new TreiD(1, 2, 3);
    TreiD b = new TreiD(10, 10, 10);
    TreiD c = new TreiD();

    Console.Write("Punctul a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Punctul b: ");
    b.show();
    Console.WriteLine();

    c = a + b;   // se aduna a si b
    Console.Write("Rezultatul a + b: ");
    c.show();
    Console.WriteLine();

    c = a + b + c;  // se aduna a, b si c
    Console.Write("Rezultatul a + b + c: ");
    c.show();
    Console.WriteLine();

    c = c - a;    // se scade a
    Console.Write("Rezultatul c - a: ");
    c.show();
```

```
    Console.WriteLine();

    c = c - b;    // se scade b
    Console.Write("Rezultatul c - b: ");
    c.show();
    Console.WriteLine();

    c = -a;    // atribuim -a lui c
    Console.Write("Rezultatul -a: ");
    c.show();
    Console.WriteLine();

    a++;    // se incrementeaza a
    Console.Write("Rezultatul a++: ");
    a.show();
  }
}
```

Rezultatul acestui program este prezentat în continuare:

```
Punctul a: 1, 2, 3

Punctul b: 10, 10, 10

Rezultatul a + b: 11, 12, 13

Rezultatul a + b + c: 22, 24, 26

Rezultatul c - a: 21, 22, 23

Rezultatul c - b: 11, 12, 13

Rezultatul -a: -1, -2, -3

Rezultatul a++: 2, 3, 4
```

După cum știți, operatorii ++ și - - au atât o formă prefixată, cât și o formă postfixată. De exemplu, atât:

```
++O;
```

cât și:

```
O++;
```

sunt forme corecte de utilizare a operatorului de incrementare. Atunci când redefiniți operatorii ++ și - -, ambele forme vor apela însă aceeași metodă. La supraîncărcare, nu există nici o modalitate de a distinge între formele prefixate și cele postfixate ale operatorilor ++ și - -.

Exerciții la minut

- Ce este supraîncărcarea operatorilor? Care este cuvântul cheie utilizat?
- Care este forma generală utilizată la supraîncărcarea operatorilor binari?
- Cu ce diferă supraîncărcarea operatorilor ++ și - - de redefinirea altor operatori?

Creșterea flexibilității

Oricum am alege o clasă și un operator, metoda operator poate fi la rândul ei supraîncărcată. De exemplu, să considerăm încă o dată clasa **TreiD**. Până în prezent, ați văzut cum se poate redefini operatorul + astfel încât să adune coordonatele unui obiect **TreiD** cu ale altuia. Acesta nu este însă singurul mod în care putem defini adunarea pentru obiectele **TreiD**. De exemplu, este uneori util să adunăm o valoare întreagă la fiecare coordonată a unui obiect **TreiD**. O astfel de operație se dovedește utilă pentru a implementa o translație a axelor. Pentru aceasta, trebuie să redefiniți operatorul + încă o dată, după cum se observă mai jos:

```
// Supraincarcam operatorul binar + pentru adunarea unei valori intregi.
 public static TreiD operator+(TreiD op1, int op2)
{
  TreiD result = new TreiD();

  result.x = op1.x + op2;
  result.y = op1.y + op2;
  result.z = op1.z + op2;

  return result;
}
```

Aceasta definește adunarea unui obiect cu un întreg.

Remarcați că al doilea parametru este de tipul **int**. Metoda de mai sus permite deci adunarea unei valori întregi la fiecare din câmpurile unui obiect **TreiD**. Operația este corectă deoarece, după cum am explicat mai devreme, la redefinirea unui operator binar, unul dintre operanzi trebuie să fie de același tip cu clasa pentru care are loc redefinirea. Celălalt operand poate fi însă de orice alt tip.

- Supraîncărcarea operatorilor permite redefinirea semnificației unor operatori în contextul unei clase create de dumneavoastră. Cuvântul cheie utilizat este **operator**.
- Forma generală utilizată la supraîncărcarea unui operator binar este următoarea:

  public static *tip-rez* **operator** *op*(*tip-param1 operand1*,
  tip-param2 operand2)
  {
    // operatii
  }
- La supraîncărcarea operatorilor ++ sau - -, metoda operator modifică de regulă valoarea operandului. Aceasta nu se întâmplă pentru ceilalți operatori.

În continuare, prezentăm versiunea clasei `TreiD` cu două definiții ale metodei +:

```
/* Se redefineste adunarea pentru obiect + obiect si pentru obiect + int. */
using System;

// O clasa cu coordonate 3D
class TreiD {
  int x, y, z;      // coordonate 3D

  public TreiD() { x = y = z = 0; }
  public TreiD(int i, int j, int k) { x = i; y = j; z = k; }

  // Redefinirea operatorului binar + pentru obiect + obiect.
  public static TreiD operator +(TreiD op1, TreiD op2)
  {
    TreiD result = new TreiD();

    /* Se aduna coordonatele celor doua puncte, intorcand result. */
    result.x = op1.x + op2.x;  // acestea sunt adunari intregi
    result.y = op1.y + op2.y;  // iar + isi pastreaza sensul
    result.z = op1.z + op2.z;  // original relativ la ele.

    return result;
  }

  // Supraincarcam operatorul binar + pentru adunarea unei valori intregi.
  public static TreiD operator+(TreiD op1, int op2)
  {
    TreiD result = new TreiD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Afiseaza coordonatele x, y, z.
  public void show() {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TreiDDemo {
  public static void Main() {
    TreiD a = new TreiD(1, 2, 3);
    TreiD b = new TreiD(10, 10, 10);
    TreiD c = new TreiD();

    Console.Write("Punctul a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Punctul b: ");
    b.show();
    Console.WriteLine();

    c = a + b;   // obiect + obiect
    Console.Write("Rezultatul a + b: ");
```



```
    c.show();
    Console.WriteLine();

    c = b + 10;  // obiect + int
    Console.Write("Rezultatul b + 10: ");
    c.show();
  }
}
```

Rezultatul afișat de acest program este prezentat mai jos:

```
Punctul a: 1, 2, 3

Punctul a: 1, 2, 3

Rezultatul a + b: 11, 12, 13

Rezultatul b + 10: 20, 20, 20
```

După cum arată și acest rezultat, atunci când operatorul + se aplică pentru două obiecte, se adună coordonatele corespunzătoare ale acestora. Dacă + se utilizează pentru un obiect și o valoare întreagă, coordonatele obiectului cresc cu acea valoare.

Chiar dacă supraîncărcarea operatorului + a adăugat clasei `TreiD` o facilitate utilă, încă mai este de lucru. Motivul este următorul: metoda `operator+(TreiD, int)` ne permite să scriem instrucțiuni de forma:

```
ob1 = ob2 + 10;
```

Din nefericire, o instrucțiune cum este următoarea:

```
ob1 = 10 + ob2;
```

nu este permisă. Problema este că instrucțiunea de mai sus are parametrul întreg în partea stângă, în timp ce metoda `operator+()` se așteaptă ca acesta să se găsească în dreapta. Pentru a permite ambele forme de instrucțiuni, trebuie să supraîncărcați operatorul + încă o dată. Noua versiune va avea primul parametru de tipul `int`, iar cel de-al doilea de tipul `TreiD`. Una din versiunile metodei `operator+()` implementează adunarea *obiect* + *întreg*, în timp ce cealaltă va implementa operația *întreg* + *obiect*. Prin redefinirea operatorului + (sau a oricărui alt operator binar) în această manieră, se permite apariția tipului predefinit atât în stânga, cât și în dreapta operatorului. Iată versiunea clasei `TreiD` care redefinește operatorul + după cum am precizat mai sus:

```
/* Se redefineste + pentru obiect + obiect, obiect + int si int + obiect. */
using System;

// O clasa cu coordonate 3D
class TreiD {
  int x, y, z;      // coordonate 3D

  public TreiD() { x = y = z = 0; }
  public TreiD(int i, int j, int k) { x = i; y = j; z = k; }

  // Redefinirea operatorului binar + pentru obiect + obiect.
  public static TreiD operator +(TreiD op1, TreiD op2) {
```

```
    TreiD result = new TreiD();

    /* Se aduna coordonatele celor doua puncte, intorcand result. */
    result.x = op1.x + op2.x;  // acestea sunt adunari intregi
    result.y = op1.y + op2.y;  // iar + isi pastreaza sensul
  result.z = op1.z + op2.z;  // original relativ la ele.

    return result;
  }

  // Redefinim operatorul binar + pentru obiect + int.
  public static TreiD operator+(TreiD op1, int op2)
  {
    TreiD result = new TreiD();

    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;

    return result;
  }

  // Redefinim operatorul binar + pentru int + obiect.
  public static TreiD operator+(int op1, TreiD op2) {
    TreiD result = new TreiD();

    result.x = op2.x + op1;
    result.y = op2.y + op1;
    result.z = op2.z + op1;

    return result;
  }

  // Afiseaza coordonatele x, y, z.
  public void show() {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TreiDDemo {
  public static void Main() {
    TreiD a = new TreiD(1, 2, 3);
    TreiD b = new TreiD(10, 10, 10);
    TreiD c = new TreiD();

    Console.Write("Punctul a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Punctul b: ");
    b.show();
    Console.WriteLine();

    c = a + b;    // obiect + obiect
    Console.Write("Rezultatul a + b: ");
    c.show();
    Console.WriteLine();
```

Adunare obiect + întreg.

Adunare întreg + obiect.


```
    c = b + 10;  // obiect + int
    Console.Write("Rezultatul b + 10: ");
    c.show();
    Console.WriteLine();

    c = 15 + b;  // int + obiect
    Console.Write("Rezultatul 15 + b: ");
    c.show();
  }
}
```

Rezultatul afişat de acest program este prezentat mai jos:

```
Punctul a: 1, 2, 3

Punctul a: 1, 2, 3

Rezultatul a + b: 11, 12, 13

Rezultatul b + 10: 20, 20, 20

Rezultatul 15 + b: 25, 25, 25
```

Supraîncărcarea operatorilor relaţionali

Operatorii relaţionali, cum ar fi == sau <, pot fi de asemenea redefiniţi într-o manieră cât se poate de clară. De regulă, un operator relaţional supraîncărcat va întoarce o valoare **true** sau **false**. Aceasta conservă modul de utilizare a acestor operatori şi permite operatorilor relaţionali supraîncărcaţi să fie utilizaţi în expresii condiţionale. Dacă alegeţi să întoarceţi un alt tip de rezultat, veţi diminua foarte mult utilitatea acestor operatori.

Prezentăm în continuare o versiune a clasei **TreiD** care redefineşte operatorii < şi >. Prin implementare, un obiect este mai mare decât un altul dacă toate cele trei coordonate sunt mai mari decât ale celuilalt obiect. Similar, un obiect este mai mic decât un alt obiect dacă toate coordonatele sale sunt mai mici decât ale celui de-al doilea obiect.

```
// Se redefinesc operatorii < si >.

using System;

// O clasa cu coordonate 3D
class TreiD {
  int x, y, z;    // coordonate 3D

  public TreiD() { x = y = z = 0; }
  public TreiD(int i, int j, int k) { x = i; y = j; z = k; }

  // Redefinirea operatorului <.
  public static bool operator <(TreiD op1, TreiD op2)
  {
    if((op1.x < op2.x) && (op1.y < op2.y) && (op1.z < op2.z))
      return true;
```

```
    else
      return false;
  }

  // Redefinirea operatorului >.
  public static bool operator >(TreiD op1, TreiD op2)
  {
    if((op1.x > op2.x) && (op1.y > op2.y) && (op1.z > op2.z))
      return true;
    else
      return false;
  }

  // Afiseaza coordonatele x, y, z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TreiDDemo {
  public static void Main() {
    TreiD a = new TreiD(5, 6, 7);
    TreiD b = new TreiD(10, 10, 10);
    TreiD c = new TreiD(1, 2, 3);

    Console.Write("Punctul a: ");
    a.show();
    Console.Write("Punctul b: ");
    b.show();
    Console.Write("Punctul c: ");
    c.show();
    Console.WriteLine();

    if(a > c) Console.WriteLine("a > c este adevarata");
    if(a < c) Console.WriteLine("a < c este adevarata");
    if(a > b) Console.WriteLine("a > b este adevarata");
    if(a < b) Console.WriteLine("a < b este adevarata");
  }
}
```

Întoarce **true** dacă op1 > op2.

Rezultatul afişat de program este prezentat mai jos:

```
Punctul a: 5, 6, 7
Punctul b: 10, 10, 10
Punctul c: 1, 2, 3

a > c este adevarata
a < b este adevarata
```

Există o restricţie importantă valabilă la redefinirea operatorilor relaţionali: aceştia trebuie redefiniţi în perechi. De exemplu, dacă redefiniţi operatorul <, atunci trebuie să redefiniţi şi > şi reciproc. Perechile de operatori sunt următoarele:

```
==    !=
<     >
<=    >=
```


Observaţie

Dacă supraîncărcaţi operatorii == şi !=, trebuie de regulă să redefiniţi metodele `Object.Equals()` şi `Object.GetHashCode()`. Aceste metode, cât şi tehnica de redefinire a lor, sunt prezentate în modulul 8.

Sfaturi şi restricţii privind supraîncărcarea operatorilor

Acţiunea unui operator supraîncărcat asupra clasei pentru care are loc redefinirea nu este neapărat să fie legată în vreun fel de utilizarea implicită a operatorului pentru tipurile predefinite din C#. Din motivele structurării şi lizibilităţii codului însă, un operator redefinit trebuie să reflecte, pe cât posibil, ideea de utilizare a operatorului original. De exemplu, operatorul + pentru clasa `TreiD` este similar din punct de vedere conceptual cu adunarea tipurilor întregi. Nu ar fi aproape de nici un folos să redefiniţi, de exemplu, într-o anumită clasă operatorul +, astfel încât acesta să se comporte similar cu ceea ce v-aţi aştepta de la operatorul /. Concluzia principală este că, deşi aveţi libertatea de a acorda unui operator supraîncărcat orice semnificaţie doriţi, pentru claritate este cel mai bine ca noua semnificaţie a operatorului să fie înrudită cu cea originală.

Există şi câteva restricţii la supraîncărcarea operatorilor. Prin această operaţie, nu puteţi modifica precedenţa nici unui operator. Nu puteţi schimba nici numărul de operanzi cerut de un operator, chiar dacă metoda operator poate ignora unul dintre operanzi. Există câţiva operatori care nu pot fi supraîncărcaţi. Cel mai important, probabil, este că nu puteţi supraîncărca nici unul dintre operatorii de atribuire, inclusiv cei compuşi, cum ar fi +=. Iată care sunt ceilalţi operatori care nu pot fi supraîncărcaţi.

```
&&       ||       []   ()   new   is
sizeof   typeof   ?    ->   .     =
```

Un ultim aspect: cuvintele cheie **true** şi **false** pot fi utilizate de asemenea ca operatori unari în scopul redefinirii. Versiunile supraîncărcate ale acestor operatori pot pune la dispoziţie determinări personalizate ale valorilor de adevăr, sau pot fi utilizate la implementarea unei logici booleene cu trei stări: adevărat, fals şi nedeterminat. Este interesant să studiaţi aceşti operatori pe cont propriu.

Exerciții la minut

- Ce trebuie să faceți pentru a permite atât împărțirea întreg/obiect, cât și pe cea inversă obiect/întreg?
- Ce tip de date întorc de regulă operatorii relaționali supraîncărcați?
- Este posibil să supraîncărcați operatorul de atribuire?

Sfatul expertului

Întrebare: Din moment ce operatori cum sunt += nu pot fi supraîncărcați, ce se întâmplă de exemplu dacă încercăm să utilizăm operatorul += pentru un obiect dintr-o clasă pentru care s-a redefinit operatorul +? Mai general, ce se întâmplă când utilizăm o atribuire compusă pentru un obiect pentru care am definit componenta operațională a atribuirii?

Răspuns: În general, dacă ați definit un anumit operator, atunci când acesta este folosit într-o atribuire compusă, se apelează metoda operator definită de dumneavoastră. Utilizarea lui += va apela deci automat versiunea metodei **operator+()** pe care ați definit-o. De exemplu, presupunând că lucrați cu clasa **TreiD**, dacă scrieți o secvență de genul:

```
TreiD a = new TreiD(1, 2, 3);
TreiD b = new TreiD(10, 10, 10);

b += a;  // se aduna a la b
```

se va invoca automat metoda **operator+()** a clasei **TreiD**, iar **b** va conține coordonatele (11, 12, 13).

Indexări

După cum știți deja, indexarea în cadrul unui tablou se realizează cu ajutorul operatorului [ ]. Este posibil să supraîncărcați operatorul [ ] pentru clasele create de dumneavoastră, dar nu prin utilizarea unei metode **operator**. Trebuie să creați în loc o *indexare*. O indexare permite unui obiect să fie indexat întocmai ca și un tablou. Principala utilitate a indexărilor apare la crearea de tablouri specializate, care sunt obiectul uneia sau mai multor restricții. Puteți însă să utilizați o indexare ori de câte ori o sintaxă apropiată de cea a tablourilor se dovedește avantajoasă. Ca și tablourile, indexările pot avea una sau mai multe dimensiuni.

- Trebuie să supraîncărcați operatorul / în două moduri, obiectul fiind o dată primul parametru și a doua oară al doilea parametru.
- Operatorii relaționali întorc de regulă tipul **bool**.
- Nu.

Vom începe cu prezentarea indexărilor unidimensionale. Indexările unidimensionale au următoarea formă generală:

```
tip-element this[int index] {
    // accesorul get.
    get {
        // se intoarce valoarea precizata de index
    }

    // accesorul set.
    set {
        // se modifica valoarea precizata prin index
    }
}
```

Mai sus, *tip-element* reprezintă tipul de bază al indexării. Fiecare element accesat de către indexare este deci de tipul *tip-element*. Acesta este similar cu tipul de bază al unui tablou obișnuit. Parametrul *index* primește valoarea indexului elementului care va fi accesat. Din punct de vedere tehnic, acest parametru nu trebuie să fie neapărat de tipul **int**, dar întrucât indexările se utilizează pentru a implementa un mecanism de acces similar tablourilor, de regulă se recurge la un tip întreg.

În corpul unei indexări sunt definiți doi *accesori*, denumiți **get** și **set**. Un accesor este similar cu o metodă, cu diferența că nu conține declarații de parametri și nici nu specifică tipul rezultatului. Accesorii sunt automat apelați la utilizarea indexării, ambii accesori primind *index* ca parametru. Dacă indexarea apare în partea stângă a unei instrucțiuni de atribuire, se apelează accesorul **set**, iar valoarea elementului specificat prin *index* se modifică. În caz contrar, se apelează accesorul **get**, întorcând valoarea care este asociată cu *index*. Metoda **set** primește de asemenea o valoare numită **value**, care este atribuită elementului cu indexul precizat.

Unul dintre avantajele utilizării unei indexări este acela că puteți controla cu exactitate modul de acces la un tablou, eliminând accesele incorecte. De exemplu, în continuare prezentăm o modalitate mai bună de implementare pentru tabloul „tolerant la erori" creat în modulul 6. Soluția utilizează o indexare, deci va permite ca tabloul să fie accesat utilizând notația obișnuită pentru tablouri.

```
// O imbunatatire a tabloului tolerant la erori prin adaugarea
// unei indexari.

using System;

class FailSoftArray {
  int [] a;    // referinta catre un tablou

  public int Length; // Length este un camp public

  public bool errflag; // indica rezultatul ultimei operatii

  // Construieste tabloul cunoscand dimensiunea.
```

```
  public FailSoftArray(int size) {
    a = new int[size];
    Length = size;
  }

  // O indexare pentru clasa FailSoftArray.
  public int this[int index] {
    // accesorul get.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index];
      } else {
        errflag = true;
        return -1;
      }
    }

    // accesorul set.
    set {
      if(ok(index)) {
        a[index] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // intoarce true daca indexul este in domeniu.
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
  }
}

// Prezinta utilizarea versiunii imbunatatite a tabloului "tolerant la erori".
class ImprovedFSDemo {
  public static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // cateva operatii esuate
    Console.WriteLine("Esuare silentioasa.");
    for(int i = 0; i < (fs.Length * 2); i++)
      fs[i] = i * 10;

    for(int i = 0; i < (fs.Length * 2); i++) {
      x = fs[i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // generam cateva erori
    Console.WriteLine("\n Esuare cu raportarea erorilor.");
    for(int i = 0; i < (fs.Length * 2); i++) {
      fs [i] = i * 10;
      if(fs.errflag)
```

O indexare pentru clasa **FailSoftArray**.

Invocarea accesorului **set** al indexării.

Invocarea accesorului **get** al indexării.


```
        Console.WriteLine("fs[" + i + "] depaseste marginile");
    }

    for(int i = 0; i < (fs.Length * 2); i++) {
      x = fs [i];
      if(! fs.errflag) Console.Write(x + " ");
      else Console.WriteLine("fs[" + i + "] depaseste marginile");
    }
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Esuare silentioasa.
0 10 20 30 40 -1 -1 -1 -1 -1

Esuare cu raportarea erorilor.
fs[5] depaseste marginile
fs[6] depaseste marginile
fs[7] depaseste marginile
fs[8] depaseste marginile
fs[9] depaseste marginile
0 10 20 30 40 fs[5] depaseste marginile
fs[6] depaseste marginile
fs[7] depaseste marginile
fs[8] depaseste marginile
fs[9] depaseste marginile
```

Acest rezultat este asemănător cu cel afișat de versiunea anterioară a programului, prezentată în modulul 6. În această versiune, indexarea este cea care evită depășirea marginilor tabloului. Să examinăm amănunțit fiecare parte a indexării. Aceasta începe cu linia:

```
public int this[int index] {
```

Această linie declară o indexare care lucrează cu elemente de tipul **int**. Indexul este transmis prin variabila **index**. Indexarea este un membru public, deci poate fi utilizată de codul situat în afara clasei. Accesorul **get** este prezentat mai jos:

```
get {
  if(ok(index)) {
    errflag = false;
    return a[index];
  } else {
    errflag = true;
    return 0;
  }
}
```

Accesorul **get** evită erorile de depășire a marginilor tabloului. Dacă indexul specificat face parte din domeniu, elementul corespunzător acelui index este întors. Dacă însă indexul depășește marginile domeniului, nu se efectuează nici o operație și nu se produce nici o depășire. În această versiune a clasei **FailSoftArray**, rezultatul fiecărei operații este conținut într-o variabilă numită **errflag**. Acest câmp poate fi examinat după fiecare operație, pentru a stabili dacă operația s-a desfășurat cu

succes sau a eşuat. (În modulul 10, veţi învăţa o modalitate mai bună de tratare a erorilor, utilizând sistemul de excepţii din C#; deocamdată, utilizarea unui indicator de eroare este o soluţie destul de bună.)

Accesorul **set** este prezentat mai jos. Şi acesta evită erorile de depăşire.

```
set {
  if(ok(index)) {
    a[index] = value;
    errflag = false;
  }
  else errflag = true;
}
```

Mai sus, dacă **index** este în domeniu, valoarea transmisă în **value** este atribuită elementului corespunzător. În caz contrar, câmpul **errflag** primeşte valoarea **true**. Reţineţi că în corpul unei metode accesor, **value** este un parametru implicit care conţine valoarea atribuită. Acesta nu trebuie (şi de fapt nici nu poate fi) declarat.

Nu este obligatoriu ca o indexare să implementeze atât **get**, cât şi **set**. Puteţi crea o indexare care poate numai citi elemente, implementând doar accesorul **get**. Reciproc, puteţi crea o indexare care poate numai scrie elemente, implementând doar accesorul **set**.

Este important să înţelegeţi că nu este neapărat ca o indexare să lucreze cu un tablou. Aceasta trebuie numai să ofere funcţionalitate care se apropie de cea a tablourilor, din punctul de vedere al utilizatorilor. De exemplu, programul următor conţine o indexare care se comportă ca un tablou accesibil numai la citire şi care conţine primele 16 puteri ale lui 2, începând cu puterea 0. Remarcaţi însă că nu există de fapt nici un tablou. Indexarea calculează valoarea asociată cu un index dat simulând existenţa unui tablou.

```
// Nu este obligatoriu ca indexarile sa lucreze cu tablouri.

using System;

class PwrOfTwo {

  /* Acces la un tablou logic care contine puterile lui 2
     de la 0 la 15. */
  public int this[int index] {
    // Calculeaza si intoarce puterea lui 2.
    get {
      if((index >= 0) && (index < 16)) return pwr(index);
      else return -1;
    }

    // nu exista accesorul set
  }

  int pwr(int p) {
    int result = 1;

    for(int i = 0; i < p; i++)
      result *= 2;

    return result;
  }
}

class UsePwrOfTwo {
  public static void Main() {
    PwrOfTwo pwr = new PwrOfTwo();

    Console.Write("Primele 8 puteri ale lui 2: ");
    for(int i = 0; i < 8; i++)
      Console.Write(pwr[i] + " ");
    Console.WriteLine();

    Console.Write("Cateva rezultate eronate: ");
    Console.Write(pwr[-1] + " " + pwr[17]);
  }
}
```

Aici, nu este indexat un tablou existent.

Rezultatul acestui program este prezentat mai jos:

```
Primele 8 puteri ale lui 2: 1 2 4 8 16 32 64 128
Cateva rezultate eronate: -1 -1
```

Remarcaţi că indexarea din clasa **PwrOfTwo** conţine un accesor **get**, dar nu dispune de accesorul **set**. După cum am explicat, aceasta înseamnă că indexarea este accesibilă numai pentru citire. Obiectele **PwrOfTwo** vor putea fi deci utilizate în partea dreaptă a instrucţiunilor de atribuire, dar nu şi în stânga. De exemplu, dacă veţi încerca să adăugaţi instrucţiunea următoare la programul anterior, acesta nu va mai funcţiona:

```
pwr[0] = 11  // nu se compileaza
```

Această instrucţiune va produce o eroare la compilare, întrucât accesorul **set** al indexării nu este definit.

Utilizarea indexărilor impune următoarea restricţie importantă: deoarece o indexare nu defineşte o zonă de memorie, valorile generate de indexări nu pot fi transmise ca parametri **ref** sau **out** ai metodelor.

Indexări multidimensionale

Aveţi de asemenea posibilitatea de a crea indexări multidimensionale. În exemplul următor, prezentăm un tablou „tolerant la erori" bidimensional. Fiţi foarte atent la modul cum se declară indexarea.

```
// Un tablou "tolerant la erori" bidimensional.
using System;

class FailSoftArray2D {
  int [,] a;   // referinta la un tablou 2D
  int rows, cols; // dimensiunile
  public int Length;  // Length este un camp public
```

```
  public bool errflag; // indica rezultatul ultimei operatii

  // Construim tabloul cunoscand dimensiunile.
  public FailSoftArray2D(int r, int c) {
    rows = r;
    cols = c;
    a = new int[rows, cols];
    Length = rows * cols;
  }

  // Indexarea pentru clasa FailSoftArray2D.
  public int this[int index1, int index2] {
    // accesorul get.
    get {
      if(ok(index1, index2)) {
        errflag = false;
        return a[index1, index2];
      } else {
        errflag = true;
        return -1;
      }
    }

    // accesorul set.
    set {
      if(ok(index1, index2)) {
        a[index1, index2] = value;
        errflag = false;
      }
      else errflag = true;
    }
  }

  // Intoarce true daca indecsii sunt in domeniu.
  private bool ok(int index1, int index2) {
    if(index1 >= 0 & index1 < rows &
      index2 >= 0 & index2 < cols)
        return true;

    return false;
  }
}

// Utilizeaza o indexare bidimensionala.
class TwoDIndexerDemo {
  public static void Main() {
    FailSoftArray2D fs = new FailSoftArray2D(3, 5);
    int x;

    // esuari silentioase.
    Console.WriteLine("Esuare silentioasa.");
    for(int i = 0; i < 6; i++)
      fs[i, i] = i * 10;

    for(int i = 0; i < 6; i++) {
      x = fs[i, i];
      if(x != -1) Console.Write(x + " ");
```

O indexare bidimensională.


```
    }
    Console.WriteLine();

    // fortam aparitia unor esecuri
    Console.WriteLine("\nEsuare cu raportarea erorilor.");
    for(int i = 0; i < 6; i++) {
      fs[i, i] = i * 10;
      if(fs.errflag)
        Console.WriteLine("fs[" + i + ", " + i +"] depaseste marginea tabloului");
      }

      for(int i = 0; i < 6; i++) {
      x = fs[i, i];
      if(! fs.errflag) Console.Write(x + " ");
      else
        Console.WriteLine("fs[" + i + ", " + i + "] depaseste marginea tabloului");
    }
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Esuare silentioasa.
0 10 20 -1 -1 -1

Esuare cu raportarea erorilor.
fs[3, 3] depaseste marginea tabloului
fs[4, 4] depaseste marginea tabloului
fs[5, 5] depaseste marginea tabloului
0 10 20 fs[3, 3] depaseste marginea tabloului
fs[4, 4] depaseste marginea tabloului
fs[5, 5] depaseste marginea tabloului
```

Exerciții la minut

- Cum se numesc accesorii unei indexări și care este rolul lor?
- Poate fi o indexare utilizată numai pentru citire?
- Puteți crea o indexare corespunzătoare unui tablou bidimensional?

Sfatul expertului

Întrebare: Indexările pot fi supraîncărcate?

Răspuns: Da. Versiunea executată va fi cea al cărei parametru formal se potrivește cel mai bine cu parametrul (sau parametrii) care se utilizează ca index.

- Accesorii se numesc **get** și **set**; ei obțin, respectiv modifică elementul curent indexat.
- Da.
- Da.

Proprietăți

Un alt tip de membri ai unei clase sunt *proprietățile.* O proprietate combină un câmp cu metodele care îl accesează. După cum am văzut și din exemplele prezentate până acum în această carte, adeseori aveți nevoie să creați un câmp disponibil celor care utilizează obiectul, menținând în același timp controlul asupra operațiilor permise asupra câmpului. De exemplu, puteți limita domeniul valorilor care pot fi atribuite câmpului respectiv. Deși acest lucru se poate realiza utilizând o variabilă privată, împreună cu metodele de acces la valoarea sa, o proprietate reprezintă o soluție mai naturală.

Proprietățile sunt entități similare cu indexările. O proprietate constă dintr-un nume și accesorii **get** și **set**. Accesorii se utilizează pentru a întoarce și respectiv modifica valoarea unei variabile. Avantajul esențial al unei proprietăți este că numele său poate fi utilizat în expresii și atribuiri la fel ca orice altă variabilă, dar în realitate accesorii **get** și **set** sunt automat invocați.

Forma generală pentru o proprietate este următoarea:

```
tip nume {
   get {
      // codul accesorului get
   }

   set {
      // codul accesorului set
   }
}
```

Mai sus, *tip* precizează tipul proprietății, ca de exemplu **int**, în timp ce *nume* este numele proprietății. După ce proprietatea a fost definită, orice utilizare a numelui *nume* va determina un apel al accesorului corespunzător. Accesorul **set** primește în mod automat un parametru numit **value**, care conține valoarea atribuită propretății.

Este esențial să înțelegeți că proprietățile nu definesc locații de memorie. Cu alte cuvinte, o proprietate doar gestionează accesul la un câmp. Proprietatea nu conține ea însăși câmpul respectiv. Câmpul trebuie specificat separat de proprietate.

În continuare, prezentăm un exemplu simplu, care definește o proprietate numită **myprop**, care este utilizată pentru accesul la câmpul **prop**. În acest caz, proprietatea permite numai atribuirea unor valori pozitive.

```
// Un exemplu simplu de proprietate.

using System;

class SimpProp {
  int prop; // camp gestionat de myprop

  public SimpProp() { prop = 0; }
```



```
  /* Aceasta este proprietatea care gestioneaza accesul la variabila instanta
     privata prop. Proprietatea nu permite decat valori pozitive. */
  public int myprop {
    get {
      return prop;
    }
    set {
      if(value >= 0) prop = value;
    }
  }
}

// Utilizeaza o proprietate.
class PropertyDemo {
  public static void Main() {
    SimpProp ob = new SimpProp();

    Console.WriteLine("Valoarea initiala a lui ob.myprop: " +
      ob.myprop);

    ob.myprop = 100; // atribuim o valoare
    Console.WriteLine("Valoarea lui ob.myprop: " + ob.myprop);

    // Nu putem atribui o valoare negativa lui prop
    Console.WriteLine("Incercam sa atribuim -10 lui ob.myprop");
    ob.myprop = -10;
    Console.WriteLine("Valoarea lui ob.myprop: " + ob.myprop);
  }
}
```

Proprietatea **myprop** controlează accesul la **prop**.

Se utilizează **myprop** la fel ca orice variabilă.

Rezultatul programului este afișat mai jos:

```
Valoarea initiala a lui ob.myprop: 0
Valoarea lui ob.myprop: 100
Incercam sa atribuim -10 lui ob.myprop
Valoarea lui ob.myprop: 100
```

Să studiem acest program cu mai multă atenție. Programul definește un câmp privat numit **prop** și o proprietate numită **myprop**, care gestionează accesul la **prop**. După cum am explicat, o proprietate nu definește o locație de memorie, ci doar controlează accesul la un câmp. Nu se poate deci concepe noțiunea de proprietate fără un câmp asociat. Mai mult, deoarece **prop** este un câmp privat, el poate fi accesat *doar* prin intermediul lui **myprop**.

Proprietatea **myprop** este specificată ca **public**, astfel încât să poată fi accesibilă pentru codul din afara clasei. Aceasta este firesc, întrucât proprietatea asigură accesul la câmpul **prop**, care este privat. Accesorul **get** întoarce pur și simplu valoarea lui **prop**. Accesorul **set** modifică valoarea lui **prop** dacă și numai dacă valoarea transmisă este pozitivă. Proprietatea **myprop** controlează astfel valorile pe care le poate lua **prop**. Acesta este principalul motiv pentru care proprietățile sunt atât de importante.

Tipul de proprietate definit de **myprop** se numește proprietate accesibilă la citire și scriere, deoarece asigură atât citirea, cât și modificarea câmpului asociat. Este însă posibil să creați proprietăți accesibile numai la citire, respectiv numai la scriere.

Pentru a crea o proprietate accesibilă numai la citire, trebuie să definiți numai accesorul **get**. Reciproc, pentru a crea o proprietate accesibilă numai la scriere, veți defini numai accesorul **set**.

Puteți utiliza o proprietate pentru a aduce încă o îmbunătățire a clasei care implementează tabloul „tolerant la erori". După cum știți, orice tablou are asociată o proprietate numită **Length**. Până acum, clasa **FailSoftArray** a utilizat un câmp public de tip întreg numit **Length** în locul acesteia. Această improvizație nu reprezintă însă o practică recomandabilă, deoarece câmpul **Length** poate primi oricând altă valoare decât lungimea tabloului „tolerant la erori". (De exemplu, un programator rău intenționat poate modifica în mod deliberat valoarea acestui câmp.) Putem remedia această situație, transformând **Length** într-o proprietate accesibilă numai la citire, ca în versiunea următoare a clasei **FailSoftArray**.

```
// Adaugam proprietatea Length la clasa FailSoftArray.

using System;

class FailSoftArray {
  int [] a;    // referinta catre un tablou
  int len;     // lungimea tabloului

  public bool errflag; // indica rezultatul ultimei operatii

  // Construieste tabloul cunoscand dimensiunea.
  public FailSoftArray(int size) {
    a = new int[size];
    len = size;
  }

  // Proprietatea Length, accesibila doar la citire.
  public int Length {
    get {
      return len;
    }
  }

  // O indexare pentru clasa FailSoftArray.
  public int this[int index] {
    // accesorul get.
    get {
      if(ok(index)) {
        errflag = false;
        return a[index];
      } else {
        errflag = true;
        return -1;
      }
    }

    // accesorul set.
    set {
      if(ok(index)) {
        a[index] = value;
```

Length este acum o proprietate.


```
        errflag = false;
      }
      else errflag = true;
    }
  }

  // intoarce true daca indexul este in domeniu.
  private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
  }
}

// Prezinta utilizarea versiunii imbunatatite a tabloului "tolerant la erori".
class ImprovedFSDemo {
  public static void Main() {
    FailSoftArray fs = new FailSoftArray(5);
    int x;

    // putem citi Length
    for(int i = 0; i < (fs.Length); i++)
      fs[i] = i * 10;

    for(int i = 0; i < (fs.Length); i++) {
      x = fs[i];
      if(x != -1) Console.Write(x + " ");
    }
    Console.WriteLine();

    // fs.Length = 10;    // Eroare, instructiune nepermisa!

  }
}
```

Length este acum o proprietate accesibilă numai la citire, care poate fi consultată, dar nu și modificată. Pentru a vă convinge de acest adevăr, încercați să eliminați semnul de comentariu de la începutul următoarei linii din program:

```
// fs.Length = 10   // Eroare, instructiune nepermisa!
```

La compilare, veți primi un mesaj de eroare, care vă informează că proprietatea **Length** poate fi doar citită.

Restricții asupra proprietăților

Proprietățile au câteva restricții importante. Mai întâi, deoarece o proprietate nu definește o locație de memorie, nu poate fi transmisă ca parametru **ref** sau **out** unei metode. În al doilea rând, proprietățile nu pot fi supraîncărcate. (Puteți totuși defini două proprietăți diferite care controlează ambele aceeași variabilă, dar aceasta este destul de neobișnuit.) În fine, o proprietate nu trebuie să altereze starea variabilei asociate la apelul accesorului **get**. Chiar dacă această regulă nu este impusă de compilator, nerespectarea ei conduce la o eroare semantică. Operația **get** nu trebuie să modifice variabila asociată.

Exerciții la minut

- Proprietățile definesc locații de memorie?
- Care este principalul avantaj adus de proprietăți?
- Poate fi transmisă o proprietate ca parametru **ref**?

Proiectul 7-1: Crearea unei clase Set

După cum am amintit la începutul acestui modul, supraîncărcarea operatorilor, indexările și proprietățile vă permit să creați clase pe deplin integrate în mediul de programare C#. Gândiți-vă la următorul aspect: definind operatorii necesari, indexările și proprietățile, puteți permite unui tip clasă să fie utilizat într-un program în același fel cu tipurile predefinite. Puteți lucra cu obiectele acelei clase prin intermediul operatorilor și indexărilor și puteți folosi obiectele clasei în expresii. Adăugarea proprietăților permite clasei să ofere o interfață compatibilă cu cea a obiectelor predefinite din C#. Pentru a demonstra crearea și integrarea unei noi clase în mediul de programare C#, acest proiect creează o clasă numită **Set**, care definește tipul mulțime.

Înainte de a începe, este important să amintim ce înseamnă în mod exact o mulțime. Pentru scopul acestui proiect, o mulțime reprezintă o colecție de elemente unice. Așadar, două elemente dintr-o mulțime dată nu pot coincide. Ordinea elementelor în cadrul unei mulțimi este irelevantă. În concluzie, mulțimea:

```
{ A, B, C }
```

coincide cu mulțimea:

```
{ A, C, B }
```

O mulțime poate fi și vidă.

Mulțimile permit un număr de operații. Cele pe care le vom implementa sunt:

- Adăugarea unui element într-o mulțime.
- Eliminarea unui element dintr-o mulțime.
- Reuniunea a două mulțimi.
- Diferența a două mulțimi.

Adăugarea unui element într-o mulțime și eliminarea unui element din mulțime sunt operații evidente. Celelalte două necesită unele precizări.

Reuniunea a două mulțimi este o mulțime care conține toate elementele existente în ambele mulțimi. (Desigur, prezența duplicatelor nu este permisă.) Vom utiliza operatorul + pentru implementarea reuniunii mulțimilor.

Diferența a două mulțimi este o mulțime care conține elementele primei mulțimi care nu fac parte din cea de-a doua mulțime. Vom utiliza operatorul – pentru a efectua diferența a două mulțimi. De exemplu, dându-se două mulțimi S1 și S2, instrucțiunea următoare elimină elementele lui S2 din S1, memorând rezultatul în S3:

```
S3 = S1 - S2;
```

- Nu, proprietatea trebuie să fie asociată unei variabile separate.
- Proprietățile permit utilizarea unei sintaxe similare cu a variabilelor la apelul metodelor accesor.
- Nu.

Dacă S2 conține toate elementele lui S1, atunci S3 va fi mulțimea vidă.

Desigur, există alte câteva operații care pot fi efectuate asupra mulțimilor. Unele dintre acestea sunt propuse ca exerciții de verificare la finalul modulului. Pe celelalte este interesant să le adăugați pe cont propriu.

Din motive de simplitate, clasa **Set** va memora mulțimi de caractere, dar aceleași idei de bază pot fi utilizate pentru a crea o clasă **Set** care poate memora orice alt tip de elemente.

Pas cu pas

1. Creați un fișier numit **SetDemo.cs**.
2. Începeți crearea clasei **Set** cu liniile următoare:

```
class Set {
char [] members; // tabloul care memoreaza multimea
int len; // numarul de elemente
```

   O mulțime este memorată în tabloul de caractere referit de variabila **members**. Numărul curent de elemente din mulțime este memorat de către **len**.

3. Adăugați clasei **Set** următorii constructori:

```
// Construieste o multime vida.
public Set() {
  len = 0;
}

// Construieste o multime fara elemente cu o dimensiune data.
public Set(int size) {
  members = new char[size]; // aloca memorie pentru multime
  len = 0; // multimea nu are deocamdata elemente
}

// Construieste o multime "clonand" o alta multime.
public Set(Set s) {
  members = new char[s.len]; // aloca memorie pentru multime
  for(int i = 0; i < s.len; i++) members[i] = s[i];
  len = s.len; // numarul de elemente
}
```

   Mulțimile pot fi construite în trei moduri. În primul rând, se poate crea o mulțime vidă. Aceasta nu conține elemente și nici nu alocă un tablou pentru acestea. Mulțimea vidă este deci doar o valoare specială. În al doilea rând, se poate crea o mulțime fără elemente cu o dimensiune dată. În fine, se poate construi o nouă mulțime „clonând" o mulțime existentă. În acest caz, cele două mulțimi vor conține aceleași elemente, dar vor fi obiecte separate.

4. Adăugați proprietatea **Length**, accesibilă numai la citire și indexarea de același tip prezentate mai jos:

```
// Implementam proprietatea Length, accesibila numai la citire.
public int Length {
  get {
    return len;
  }
}

// Implementam indexarea, accesibila numai la citire.
public int this[int idx] {
  get {
```

```
    if(idx >= 0 & idex < len) return members[idx];
    else return (char)0;
  }
}
```

Funcționarea proprietății **Length** este evidentă. Indexarea întoarce elementul din mulțime corespunzător indexului primit. Pentru a evita depășirea tabloului, se efectuează o verificare a marginilor. Dacă indexul este incorect, se întoarce caracterul nul.

5. Adăugați metoda **find()** prezentată mai jos. Această metodă stabilește dacă elementul specificat prin **ch** face parte din mulțime. Metoda întoarce indexul elementului găsit, sau –1 dacă acesta nu face parte din mulțime.

```
/* Stabileste daca un element face parte dintr-o multime.
   Intoarce indexul elementului, sau –1 daca acesta nu este gasit. */
int find(char ch) {
  int i;

  for(i = 0; i < len; i++)
    if(members[i] == ch) return i;

  return –1;
}
```

6. Începeți să adăugați operatorii, primul fiind cel de reuniune. Pentru aceasta, supraîncărcați operatorul + pentru obiecte de tipul **Set**, ca mai jos. Această versiune adaugă un element la o mulțime.

```
// Adauga un element la o multime.
public static Set operator+(Set ob, char ch) {
  Set newset = new Set(ob.len + 1); // construim o multime
                                    // care are cu un element in plus

  // copiem elementele
  for(int i = 0; i < ob.len; i++)
    newset.members[i] = ob.members[i];

  // stabilim valoarea lui len
  newset.len = ob.len;

  // verificam daca elementul exista deja in multime
  if(ob.find(ch) == -1)  {  // nu am gasit elementul, adaugam
    // adauga elementul la noua multime
    newset.members[newset.len] = ch;
    newset.len++;
  }
  return newset;    // intoarce multimea actualizata
}
```

Această metodă se pretează la o examinare ceva mai amănunțită. Mai întâi, ea creează o nouă mulțime în care va memora conținutul mulțimii inițiale referite de **ob** și, eventual, noul element **ch**. Remarcați că noua mulțime creată are cu un element mai mult decât **ob**, pentru a putea memora și noul element. În continuare, se copiază elementele din mulțimea inițială în noua mulțime, iar lungimea noii mulțimi primește valoarea lungimii mulțimii inițiale. Aceasta deoarece **ch** va fi adăugat la noua mulțime numai dacă nu face parte din mulțimea inițială. Variabila **len** este deci inițializată cu lungimea mulțimii inițiale, fiind ulterior incrementată cu o unitate, dacă noul element este adăugat. În continuare, se apelează metoda **find()** pentru a determina dacă **ch** face parte din mulțime. Dacă nu face parte, se adaugă **ch** la mulțime și se actualizează valoarea lui **len**. Indiferent de situație, metoda întoarce **newset**. Mulțimea inițială nu este deci modificată în urma acestei operații.

7. În continuare, supraîncărcați operatorul – astfel încât să elimine un element din mulțime, după cum se observă mai jos:

```
// Elimina un element din multime.
public static Set operator-(Set ob, char ch) {
  Set newset = new Set();
  int i = ob.find(ch);   // i va fi –1 daca elementul nu exista

  // copiem si compactam elementele ramase dupa eliminare
  for(int j = 0; j < ob.len; j++)
    if(j != i) newset = newset + ob.members[i];

  return newset;
}
```

Metoda creează mai întâi o mulțime vidă. În al doilea rând, se apelează **find()** pentru a determina dacă **ch** face parte din mulțimea inițială. Amintiți-vă că **find()** întoarce –1 dacă **ch** nu face parte din mulțime. În continuare, se adaugă elementele din mulțimea inițială la noua mulțime, cu excepția aceluia al cărui index este egal cu cel întors de **find()**. Mulțimea rezultată conține deci toate elementele mulțimii inițiale, mai puțin **ch**. Dacă **ch** nu face parte din mulțimea inițială, cele două mulțimi sunt egale.

8. Redefiniți operatorii + și – încă o dată, la fel ca mai jos. Aceste versiuni implementează reuniunea și diferența mulțimilor.

```
// Reuniunea multimilor.
public static Set operator+(Set ob1, Set ob2) {
  Set newset = new Set(ob1);   // "clonam" prima multime

  // adaugam pe rand elementele din cea de-a doua multime
  for(int i = 0; i < ob2.len; i++)
    newset = newset + ob2[i];

  return newset; // intoarce multimea actualizata
}

// Diferenta multimilor.
  public static Set operator-(Set ob1, Set ob2) {
  Set newset = new Set(ob1);   // "clonam" prima multime

  // eliminam pe rand elementele celei de-a doua multimi
  for(int i = 0; i < ob2.len; i++)
    newset = newset - ob2[i];

  return newset; // intoarce multimea actualizata
}
```

După cum observați, aceste metode utilizează versiunile definite anterior ale operatorilor + și – ca metode ajutătoare. În cazul reuniunii de mulțimi, se creează o nouă mulțime, care conține elementele din prima mulțime. Se adaugă apoi pe rând elementele din cea de-a doua mulțime. Deoarece operația + adaugă un element numai dacă acesta nu face deja parte din mulțime, mulțimea rezultată este chiar reuniunea (fără duplicate) celor două mulțimi date. Operatorul de diferență elimină elementele pe care le găsește din prima mulțime.

9. Iată codul complet al clasei `Set`, împreună cu clasa `SetDemo`, care o utilizează:

```
/*
   Proiectul 7-1

   O clasa care implementeaza o multime de caractere.
*/
using System;

class Set {
  char [] members; // tabloul care memoreaza multimea
  int len; // numarul de elemente

  // Construieste o multime vida.
  public Set() {
    len = 0;
  }

  // Construieste o multime fara elemente cu o dimensiune data.
  public Set(int size) {
    members = new char[size];   // aloca memorie pentru multime
    len = 0; // multimea nu are deocamdata elemente
  }

  // Construieste o multime "clonand" o alta multime.
  public Set(Set s) {
    members = new char[s.len];  // aloca memorie pentru multime
    for(int i = 0; i < s.len; i++) members[i] = s[i];
    len = s.len; // numarul de elemente
  }

  // Implementam proprietatea Length, accesibila numai la citire.
  public int Length {
    get {
      return len;
    }
  }

  // Implementam indexarea, accesibila numai la citire.
  public int this[int idx] {
    get {
      if(idx >= 0 & index < len) return members[idx];
      else return (char)0;
    }
  }

  /* Stabileste daca un element face parte dintr-o multime.
     Intoarce indexul elementului, sau -1 daca acesta nu este gasit. */
  int find(char ch) {
    int i;

    for(i = 0; i < len; i++)
      if(members[i] == ch) return i;

    return -1;
  }
```



```
  // Adauga un element la o multime.
  public static Set operator+(Set ob, char ch) {
    Set newset = new Set(ob.len + 1);    // construim o multime
                                         // care are cu un element in plus

    // copiem elementele
    for(int i = 0; i < ob.len; i++)
      newset.members[i] = ob.members[i];

    // stabilim valoarea lui len
    newset.len = ob.len;

    // verificam daca elementul exista deja in multime
    if(ob.find(ch) == -1)  {  // nu am gasit elementul, adaugam
      // adauga elementul la noua multime
      newset.members[newset.len] = ch;
      newset.len++;
    }
    return newset;   // intoarce multimea actualizata
  }

  // Elimina un element din multime.
  public static Set operator-(Set ob, char ch) {
    Set newset = new Set();
    int i = ob.find(ch); // i va fi -1 daca elementul nu exista

    // copiem si compactam elementele ramase dupa eliminare
    for(int j = 0; j < ob.len; j++)
      if(j != i) newset = newset + ob.members[i];

    return newset;
  }

  // Reuniunea multimilor.
  public static Set operator+(Set ob1, Set ob2) {
    Set newset = new Set(ob1);   // "clonam" prima multime

    // adaugam pe rand elementele din cea de-a doua multime
    for(int i = 0; i < ob2.len; i++)
      newset = newset + ob2[i];

    return newset;// intoarce multimea actualizata
  }

  // Diferenta multimilor.
  public static Set operator-(Set ob1, Set ob2) {
    Set newset = new Set(ob1);   // "clonam" prima multime

    // eliminam pe rand elementele celei de-a doua multimi
    for(int i = 0; i < ob2.len; i++)
      newset = newset - ob2[i];

    return newset;// intoarce multimea actualizata
  }
}

// Utilizeaza clasa Set.
```

```
class SetDemo {
  public static void Main() {
    // construim trei multimi vide
    Set s1 = new Set();
    Set s2 = new Set();
    Set s3 = new Set();

    s1 = s1 + 'A';
    s1 = s1 + 'B';
    s1 = s1 + 'C';

    Console.Write("s1 dupa adaugarea A B C: ");
    for(int i = 0; i < s1.Length; i++)
    Console.Write(s1[i] + " ");
    Console.WriteLine();

    s1 = s1 - 'B';
    Console.Write("s1 dupa s1 = s1 - 'B': ");
    for(int i = 0; i < s1.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine();

    s1 = s1 - 'A';
    Console.Write("s1 dupa s1 = s1 - 'A': ");
    for(int i = 0; i < s1.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine();

    s1 = s1 - 'C';
    Console.Write("s1 dupa s1 = s1 - 'C': ");
    for(int i = 0; i < s1.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine("\n");

    s1 = s1 + 'A';
    s1 = s1 + 'B';
    s1 = s1 + 'C';
    Console.Write("s1 dupa adaugarea A B C: ");
    for(int i = 0; i < s1.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine();

    s2 = s2 + 'A';
    s2 = s2 + 'X';
    s2 = s2 + 'W';
    Console.Write("s2 dupa adaugarea A X W: ");
    for(int i = 0; i < s2.Length; i++)
      Console.Write(s2[i] + " ");
    Console.WriteLine();

    s3 = s1 + s2;
    Console.Write("s3 dupa s3 = s1 + s2: ");
    for(int i = 0; i < s3.Length; i++)
      Console.Write(s3[i] + " ");
    Console.WriteLine();

    s3 = s3 - s1;
```



```
    Console.Write("s3 dupa s3 = s3 - s1: ");
    for(int i = 0; i < s3.Length; i++)
      Console.Write(s3[i] + " ");
    Console.WriteLine("\n");

    s2 = s2 - s2;   // "stergem" s2
    s2 = s2 + 'C';  // adaugam ABC in ordine inversa
    s2 = s2 + 'B';
    s2 = s2 + 'A';

    Console.Write("s1 este acum: ");
    for(int i = 0; i < s1.Length; i++)
      Console.Write(s1[i] + " ");
    Console.WriteLine();

    Console.Write("s2 este acum: ");
    for(int i = 0; i < s2.Length; i++)
      Console.Write(s2[i] + " ");
    Console.WriteLine();

    Console.Write("s3 este acum: ");
    for(int i = 0; i < s3.Length; i++)
      Console.Write(s3[i] + " ");
    Console.WriteLine();
  }
}
```

Acest program afișează rezultatul prezentat mai jos:

```
s1 dupa adaugarea A B C: A B C
s1 dupa s1 = s1 - 'B': A C
s1 dupa s1 = s1 - 'A': C
s1 dupa s1 = s1 - 'C':

s1 dupa adaugarea A B C: A B C
s2 dupa adaugarea A X W: A X W
s3 dupa s3 = s1 + s2: A B C X W
s3 dupa s3 = s3 - s1: X W

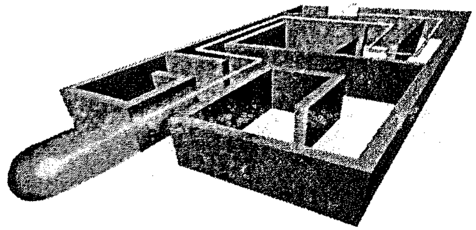
s1 este acum: A B C
s2 este acum: C B A
s3 este acum: X W
```

Verificarea cunoștințelor

1. Prezentați forma generală utilizată la supraîncărcarea unui operator unar. Ce tip trebuie să aibă parametrul metodei operator?
2. Ce trebuie să faceți pentru a permite operații care implică un tip clasă și un tip predefinit?
3. Poate fi operatorul ternar ? supraîncărcat? Puteți modifica precedența unui operator?
4. Ce este o indexare? Prezentați forma generală a unei indexări.
5. În cadrul unei indexări, ce rol au funcțiile accesor get și set?

6. Ce este o proprietate? Prezentaţi forma generală a unei proprietăţi.
7. O proprietate defineşte o locaţie de memorare? Dacă nu, unde îşi memorează o proprietate valoarea?
8. Pot fi transmise proprietăţile ca parametri `ref` sau `out`?
9. Pentru clasa `Set` dezvoltată în cadrul proiectului 7-1, definiţi operatorii < şi >, astfel încât să stabilească dacă o mulţime dată este inclusă în altă mulţime sau include o altă mulţime. Operatorul < va întoarce `true` dacă prima mulţime este o submulţime a celei de-a doua şi `false` în caz contrar. Operatorul > va întoarce `true` dacă prima mulţime o include pe a doua şi `false` în caz contrar.
10. Pentru clasa `Set`, definiţi operatorul & astfel încât să determine intersecţia a două mulţimi date.
11. Pe cont propriu, încercaţi să adăugaţi şi alţi operatori la clasa `Set`. De exemplu, încercaţi să definiţi | astfel încât să determine *diferenţa simetrică* a două mulţimi. (Diferenţa simetrică va conţine toate elementele necomune celor două mulţimi.)

Modulul 8
Moştenirea



Scopurile acestui modul

- **Noţiuni fundamentale despre moştenire**
- **Utilizarea accesului protejat**
- **Apelul constructorilor clasei de bază**
- **Utilizarea cuvântului cheie `base`**
- **Crearea unei ierarhii de clase cu mai multe niveluri**
- **Atribuirea unor instanţe ale claselor derivate variabilelor de tipul clasei de bază**
- **Crearea metodelor virtuale**
- **Utilizarea claselor abstracte**
- **Utilizarea cuvântului cheie `sealed`**
- **Studiul clasei `object`**

Moştenirea este unul din cele trei principii fundamentale ale programării orientate spre obiecte, deoarece permite crearea clasificărilor ierarhizate. Utilizând moştenirea, puteţi crea o clasă generală, care defineşte caracteristici comune unei mulţimi de elemente înrudite. Această clasă poate fi apoi moştenită de alte clase, mai specifice, fiecare adăugând numai caracteristicile care o identifică în mod unic.

În terminologia C#, o clasă care este moştenită poartă numele de *clasă de bază*. Clasa care moşteneşte anumite caracteristici se numeşte *clasă derivată*. O clasă derivată este deci o versiune specializată a unei clase de bază. Ea moşteneşte toate variabilele, metodele, proprietăţile şi indexările definite în clasa de bază, la care adaugă elementele sale proprii, unice.

Noţiuni de bază despre moştenire

Limbajul C# implementează conceptul de moştenire permiţând unei clase să încorporeze o altă clasă în declaraţia sa. Aceasta se realizează precizând clasa de bază la declararea clasei derivate. Vom începe cu un exemplu simplu, care prezintă câteva dintre facilităţile esenţiale oferite de moştenire. Programul următor creează o clasă de bază numită `Forma2D` care memorează lăţimea şi înălţimea unui obiect bidimensional, după care creează o clasă derivată numită `Triunghi`. Remarcaţi cu atenţie modul în care este declarată clasa `Triunghi`.

```
// O ierarhie simpla de clase.
using System;

// O clasa care reprezinta obiecte bidimensionale.
class Forma2D {
  public double width;
  public double height;

  public void showDim() {
    Console.WriteLine("Latimea si inaltimea sunt " +
                      width + " si " + height);
  }
}

// clasa Triunghi deriva din clasa Forma2D.
class Triunghi : Forma2D {
  public string style;

  public double aria() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
  }
}

class Forme {
  public static void Main() {
    Triunghi t1 = new Triunghi();
    Triunghi t2 = new Triunghi();

    t1.width = 4.0;
    t1.height = 4.0;
    t1.style = "isoscel";

    t2.width = 8.0;
    t2.height = 12.0;
    t2.style = "dreptunghic";

    Console.WriteLine("Informatii despre t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Aria triunghiului este " + t1.aria());

    Console.WriteLine();

    Console.WriteLine("Informatii despre t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Aria triunghiului este " + t2.aria());
  }
}
```

Clasa **Triunghi** moştenește clasa **Forma2D**. Rețineți sintaxa.

Clasa **Triunghi** poate referi membrii clasei **Forma2D** la fel ca pe membrii proprii.

Toţi membrii clasei **Triunghi** sunt accesibili instanţelor clasei, inclusiv cei moşteniţi de la **Forma2D**.

Rezultatul acestui program este prezentat mai jos:


```
Informatii despre t1:
Triunghiul este isoscel
Latimea si inaltimea sunt 4 si 4
Aria triunghiului este 8

Informatii despre t2:
Triunghiul este dreptunghic
Latimea si inaltimea sunt 8 si 12
Aria triunghiului este 48
```

În program, clasa **Forma2D** defineşte atributele unei forme „generice" bidimensionale, care poate fi un pătrat, un dreptunghi, un triunghi etc. Clasa **Triunghi** creează un tip specific de formă bidimensională, care în acest caz este un triunghi. Clasa **Triunghi** include întreaga clasă **Forma2D**, la care adaugă câmpul **style** şi metodele **aria()** şi **showStyle()**. Câmpul **style** conţine o descriere a tipului triunghiului, **aria()** calculează şi întoarce aria triunghiului, iar **showStyle()** afişează tipul triunghiului.

Remarcaţi sintaxa folosită pentru moştenirea unei clase de bază. Numele clasei de bază urmează după numele clasei derivate, fiind separat prin caracterul două puncte (:). Sintaxa pentru moştenirea unei clase este extrem de simplă şi de uşor de utilizat.

Deoarece clasa **Triunghi** include toţi membrii clasei de bază **Forma2D**, poate accesa direct **width** şi **height** în corpul metodei **aria()**. De asemenea, în **Main()**, obiectele **t1** şi **t2** pot referi câmpurile **width** şi **height** direct, ca şi cum acestea ar fi membre ale clasei **Triunghi**.

Chiar dacă **Forma2D** este clasa de bază pentru **Triunghi**, aceasta este de asemenea o clasă de sine stătătoare. Faptul că este clasa de bază din care derivă o altă clasă nu o împiedică să fie utilizată în mod independent. De exemplu, secvenţa de mai jos este perfect corectă:

```
Forma2D shape = new Forma2D();

shape.width = 10;
shape.height = 20;

shape.showDim();
```

Desigur, o instanţă a clasei **Forma2D** nu cunoaşte şi nici nu are acces la nici una din clasele derivate din **Forma2D**.

Forma generală a unei declaraţii **class** care moşteneşte o clasă de bază este prezentată mai jos:

class *nume-clasa-derivata*: *nume-clasa-baza* {
    // corpul clasei derivate
}

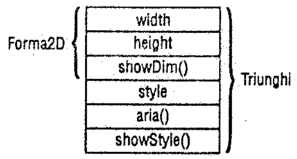


Figura 8-1 O schemă conceptuală a clasei **Triunghi**

Pentru orice clasă derivată pe care o creați, puteți specifica numai o singură clasă de bază. Limbajul C# nu permite ca o aceeași clasă derivată să moștenească mai multe clase de bază. (Aceasta diferă față de C++, în care este posibilă moștenirea mai multor clase de bază. Luați cât se poate de serios în considerare acest aspect atunci când rescrieți programe C++ în C#.) Puteți însă să creați o ierarhie de clase, în care o clasă derivată să devină clasa de bază din care sunt derivate alte clase. Evident, nici o clasă nu poate să fie clasă de bază pentru ea însăși.

Unul din avantajele esențiale ale moștenirii este că, după ce ați creat o clasă de bază care definește atributele comune pentru o mulțime de obiecte, aceasta poate fi utilizată pentru a deriva oricâte alte clase mai particulare. Fiecare clasă derivată își poate stabili cu exactitate propria clasificare. De exemplu, iată o altă clasă derivată din **Forma2D** care încapsulează dreptunghiuri:

```
// O clasa derivata din Forma2D pentru dreptunghiuri.
class Dreptunghi : Forma2D {
  public bool estePatrat() {
    if(width == height) return true;
    return false;
  }

  public double aria() {
    return width * height;
  }
}
```

Clasa **Dreptunghi** include **Forma2D**, la care adaugă metodele **estePatrat()**, care determină dacă dreptunghiul este pătrat, și **aria()**, care calculează aria dreptunghiului.

Accesul la membri și moștenirea

După cum ați învățat în modulul 6, membrii unei clase sunt adeseori declarați ca privați pentru a evita utilizarea neautorizată sau modificarea incorectă. Moștenirea unei clase *nu anulează* restricțiile impuse de accesul privat. În consecință, deși o clasă derivată include toți membrii clasei de bază, nu are acces la membrii privați ai clasei de bază. De exemplu, dacă, după cum se vede mai jos, câmpurile **width** și **height** sunt făcute private, clasa derivată **Triunghi** nu le va mai putea accesa.


```
// Membrii privati nu se mostenesc.

// Acest exemplu nu va putea fi compilat.
using System;

// o clasa care reprezinta obiecte bidimensionale.
class Forma2D {
  double width;    // este acum privat
  double height;   // este acum privat

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width
                      + " si " + height);
  }
}

// clasa Triunghi este derivata din Forma2D.
class Triunghi : Forma2D {
  public string style;

  public double aria() {
    return width * height / 2; // eroare, membri inaccesibili
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
  }
}
```

Nu avem acces la membrii privați ai clasei de bază.

Clasa **Triunghi** nu se poate compila, întrucât referirea directă a câmpurilor **width** și **height** în corpul metodei **aria()** determină o violare a accesului. Deoarece **width** și **height** sunt acum private, ele sunt accesibile numai celorlalți membri ai clasei lor. Clasele derivate nu îi pot accesa.

Rețineți: membrii privați ai claselor se pot utiliza numai în clasa în care sunt definiți. Aceștia nu sunt accesibili pentru codul din afara clasei lor, inclusiv pentru clasele derivate.

La o primă analiză, puteți crede că interzicerea accesului la membrii privați pentru clasele derivate este o restricție mult prea severă, deoarece împedică utilizarea membrilor privați și în situații în care aceștia ar fi necesari. Aceasta nu este însă adevărat, limbajul C# oferind mai multe soluții de a depăși acest inconvenient. O primă posibilitate este utilizarea membrilor **protected**, care este prezentată în secțiunea următoare. O altă soluție este utilizarea unor proprietăți publice, care să asigure accesul la membrii privați. După cum ați văzut în modulele precedente, programatorii în C# oferă de regulă accesul la membrii privați ai unei clase prin intermediul metodelor sau prin transformarea acestora în proprietăți. Iată o variantă a clasei **Forma2D** care transformă câmpurile **width** și **height** în proprietăți:

```
// Utilizarea proprietatilor pentru accesul la membrii privati.
using System;
```

```
// o clasa care reprezinta obiecte bidimensionale.
class Forma2D {
  double pri_width;   // este acum privat
  double pri_height;  // este acum privat

  // Proprietati pentru accesul la width si height.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width
                      + " si " + height);
  }
}

// clasa Triunghi este derivata din Forma2D.
class Triunghi : Forma2D {
  public string style;

  public double aria() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
  }
}

class Forme2 {
  public static void Main() {
    Triunghi t1 = new Triunghi();
    Triunghi t2 = new Triunghi();

    t1.width = 4.0;
    t1.height = 4.0;
    t1.style = "isoscel";

    t2.width = 8.0;
    t2.height = 12.0;
    t2.style = "dreptunghic";

    Console.WriteLine("Informatii despre t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Aria triunghiului este " + t1.aria());

    Console.WriteLine();

    Console.WriteLine("Informatii despre t2: ");
    t2.showStyle();
```

Proprietățile **width** și **height**.

Utilizarea lui **width** și **height** prin intermediul proprietăților este acum corectă.


```
    t2.showDim();
    Console.WriteLine("Aria triunghiului este " + t2.aria());
  }
}
```

Sfatul expertului

Întrebare: În discuțiile despre programarea în Java, am auzit termenii de „superclasă" și „subclasă". Au aceşti termeni vreo semnificație în C#?

Răspuns: Ceea ce se numeşte în Java superclasă, în C# este cunoscut sub denumirea de clasă de bază. Termenul de subclasă din Java se traduce în C# prin clasă derivată. Veți auzi în mod obişnuit cele două denumiri pentru clase în ambele limbaje, dar în această carte vom utiliza numai terminologia standard din C#. Ca fapt divers, şi în C++ se întrebuințează tot termenii de clasă de bază şi clasă derivată.

Exerciții la minut

- Cum se poate moşteni o clasă de bază într-o clasă derivată?
- O clasă derivată conține membrii clasei de bază?
- Poate o clasă derivată să acceseze membrii privați ai clasei sale de bază?

Utilizarea accesului protejat

După cum am explicat în secțiunea anterioară, membrii privați ai unei clase nu sunt accesibili în clasele derivate. De aici, s-ar părea că, rezultă că, dacă doriți ca o clasă derivată să aibă acces la anumiți membri ai clasei de bază, aceştia trebuie declarați ca publici. Declararea membrilor ca publici îi face însă accesibili în tot restul codului, ceea ce nu este întotdeauna de dorit. Din fericire, acest raționament este greşit, deoarece limbajul C# vă permite să definiți *membri protejați*. Membrii protejați sunt publici în cadrul unei ierarhii de clase, dar privați în afara acelei ierarhii.

Membrii protejați se pot crea utilizând modificatorul de acces **protected**. Atunci când un membru al unei clase este declarat ca **protected**, acel membru este, cu o excepție notabilă, privat. Excepția se produce atunci când membrul protejat este moştenit. În acest caz, membrul protejat al clasei de bază devine un membru protejat al clasei derivate, fiind deci accesibil în clasa derivată. Aşadar, utilizând

- Clasa de bază este specificată după numele clasei derivate, de care se separă prin caracterul două puncte (:).
- Da.
- Nu.

protected, puteţi crea membri privaţi ai unei clase, care sunt însă moşteniţi şi pot fi accesaţi în clasele derivate.

Iată un exemplu simplu care utilizează **protected**:

```
// Prezinta accesul protejat.
using System;

class B {
  protected int i, j;  // membri privati ai clasei B, accesibili insa in clasa D

  public void set(int a, int b) {
    i = a;
    j = b;
  }

  public void show() {
    Console.WriteLine(i + " " + j);
  }
}

class D : B {
  int k;  // membru privat

  // clasa D poate accesa campurile i si j din clasa B
  public void setk() {
    k = i * j;
  }

  public void showk() {
    Console.WriteLine(k);
  }
}

class ProtectedDemo {
  public static void Main() {
    D ob = new D();

    ob.set(2, 3); // corect, set() este mostenita
    ob.show();  // corect, show() este mostenita

    ob.setk(); // corect, setk() este in clasa D
    ob.showk(); // corect, showk() este in clasa D
  }
}
```

Câmpurile **i** şi **j** sunt **protected**.

Clasa **D** poate accesa **i** şi **j** deoarece sunt membri protejaţi, nu privaţi.

În acest exemplu, deoarece clasa **B** este moştenită de către **D**, iar **i** şi **j** sunt declaraţi ca **protected** în **B**, metoda **setk()** poate accesa aceste câmpuri. Dacă **i** şi **j** ar fi fost declaraţi ca membri privaţi în **B**, clasa **D** nu ar mai fi avut acces la ei, iar programul nu s-ar mai fi compilat.

Ca şi **public** sau **private**, statutul de **protected** al unui anumit membru se păstrează indiferent de numărul nivelurilor de moştenire. Prin urmare, dacă o clasă derivată este la rândul ei moştenită de o a doua clasă derivată, toţi membrii protejaţi ai clasei de bază iniţiale moşteniţi de prima clasă derivată sunt de asemenea moşteniţi şi rămân protejaţi şi în a doua clasă derivată.

Constructorii şi moştenirea

Într-o ierarhie, atât clasele de bază, cât şi cele derivate pot dispune de constructori proprii. Aceasta ridică însă o întrebare importantă: ce constructor se utilizează pentru a construi un obiect aparţinând unei clase derivate? Cel din clasa de bază, cel din clasa derivată, sau ambii? Răspunsul este următorul: constructorul clasei de bază construieşte porţiunea obiectului care aparţine de clasa de bază, iar constructorul clasei derivate construieşte porţiunea care aparţine strict de clasa derivată. Este logic să fie aşa, deoarece clasa de bază nu cunoaşte elementele specifice clasei derivate şi nici nu are acces la acestea. Construirea acestora trebuie să fie deci separată. În exemplele anterioare, am utilizat constructorii impliciţi creaţi în mod automat de compilatorul de C#, deci nu ne-am preocupat de această problemă. În practică însă, majoritatea claselor dispun de constructori proprii. În continuare veţi învăţa cum să rezolvaţi astfel de probleme.

Atunci când numai clasa derivată defineşte un constructor, procesul este evident: se construieşte numai obiectul aparţinând clasei derivate. Porţiunea care ţine de clasa de bază este construită automat, utilizând constructorul implicit. De exemplu, iată o versiune rescrisă a clasei **Triunghi**, care defineşte un constructor. Membrul **style** este acum privat, deoarece este iniţializat cu ajutorul constructorului.

```
// Adaugarea unui constructor la clasa Triunghi.
using System;

// o clasa care reprezinta obiecte bidimensionale.
class Forma2D {
  double pri_width;  // privat
  double pri_height; // privat

  // Proprietati pentru accesul la width si height.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width
                      + " si " + height);
  }
}

// clasa Triunghi este derivata din Forma2D.
class Triunghi : Forma2D {
  string style;   // privat

  // Constructor
```

```
  public Triunghi(string s, double w, double h) {
    width = w;   // initializam membrii clasei de baza
    height = h;  // initializam membrii clasei de baza

    style = s;   // initializam membrul clasei derivate
  }

  public double aria() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
  }
}

class Forme3 {
  public static void Main() {
    Triunghi t1 = new Triunghi("isoscel", 4.0, 4.0);
    Triunghi t2 = new Triunghi("dreptunghic", 8.0, 12.0);

    Console.WriteLine("Informatii despre t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Aria triunghiului este " + t1.aria());

    Console.WriteLine();

    Console.WriteLine("Informatii despre t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Aria triunghiului este " + t2.aria());
  }
}
```

Construirea porțiunii **Forma2D** a obiectului.

În acest exemplu, constructorul clasei **Triunghi** inițializează și membrii moșteniți de la clasa **Forma2D** împreună cu câmpul propriu **style**.

Atunci când atât clasa de bază, cât și clasa derivată definesc constructori, procesul este ceva mai complicat, deoarece ambii constructori trebuie să fie executați. În acest caz, trebuie să utilizați un alt cuvânt cheie din C#: **base**, care are două semnificații. Într-o primă instanță, **base** apelează constructorul clasei de bază. Cea de-a doua semnificație se utilizează pentru accesul la un membru al clasei de bază care a fost ascuns de un membru al clasei derivate. În continuare, vom examina prima dintre cele două semnificații.

Apelul constructorilor clasei de bază

O clasă derivată poate apela un constructor definit în clasa sa de bază utilizând o formă extinsă a constructorului propriu și cuvântul cheie **base**. Forma generală a declarației extinse este prezentată mai jos:

constructor-derivat(*lista-param*): **base**(*lista-param-efectivi*) {
  // corpul constructorului
}

Mai sus, *lista-param-efectivi* specifică parametrii efectivi necesari la apelul constructorului clasei de bază. Remarcați poziția caracterului două puncte (:).

Pentru a vedea **base** într-un exemplu, să considerăm versiunea clasei **Forma2D** din următorul program. Aceasta definește un constructor care inițializează proprietățile **width** și **height**.

```
// Adaugam constructori la clasa Forma2D.
using System;

// o clasa care reprezinta obiecte bidimensionale.
class Forma2D {
  double pri_width;   // privat
  double pri_height;  // privat

  // Constructor pentru clasa Forma2D.
  public Forma2D(double w, double h) {
    width = w;
    height = h;
  }

  // Proprietati pentru accesul la width si height.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width
                      + " si " + height);
  }
}

// clasa Triunghi este derivata din Forma2D.
class Triunghi : Forma2D {
  string style;   // privat

  // Apelul constructorului clasei de baza
  public Triunghi(string s, double w, double h) : base(w, h) {
    style = s;
  }

  public double aria() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
```

Utilizăm **base** pentru a executa constructorul clasei **Forma2D**.

```
    }
}

class Forme4 {
  public static void Main() {
    Triunghi t1 = new Triunghi("isoscel", 4.0, 4.0);
    Triunghi t2 = new Triunghi("dreptunghic", 8.0, 12.0);

    Console.WriteLine("Informatii despre t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Aria triunghiului este " + t1.aria());

    Console.WriteLine();

    Console.WriteLine("Informatii despre t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Aria triunghiului este " + t2.aria());
  }
}
```

În acest program, constructorul **Triunghi()** apelează **base** cu parametrii **w** şi **h**. Aceasta determină apelul constructorului **Forma2D()**, care iniţializează **width** şi **height** cu valorile primite. Constructorul **Triunghi()** nu se mai preocupă şi cu iniţializarea membrilor clasei de bază. Acesta nu trebuie să iniţializeze decât câmpul specific clasei derivate: **style**. Constructorul clasei **Forma2D** capătă astfel libertate deplină la construirea sub-instanţelor. Mai mult, clasa **Forma2D** poate adăuga funcţionalitate pe care clasele derivate existente să nu o cunoască deloc, evitând astfel nefuncţionarea codului deja existent.

Utilizând **base**, putem apela toate formele de constructori definite în clasa de bază. Constructorul executat este cel care se potriveşte cu parametrii efectivi. De exemplu, în continuare prezentăm versiunile extinse ale claselor **Forma2D** şi **Triunghi**, care conţin constructorii impliciţi şi constructori cu un parametru.

```
// Mai multi constructori in clasa Forma2D.
using System;

class Forma2D {
  double pri_width;   // privat
  double pri_height;  // privat

  // Constructorul implicit.
  public Forma2D() {
    width = height = 0.0;
  }

  // Constructorul clasei Forma2D.
  public Forma2D(double w, double h) {
    width = w;
    height = h;
  }
```



```
  // Construim un obiect cu latimea si inaltimea egale.
  public Forma2D(double x) {
    width = height = x;
  }

  // Proprietati pentru accesul la width si height.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width
                      + " si " + height);
  }
}

// clasa Triunghi este derivata din Forma2D.
class Triunghi : Forma2D {
  string style;    // privat

  /* Un constructor implicit. Acesta invoca automat
     constructorul implicit al clasei Forma2D. */
  public Triunghi() {
    style = "null";
  }

  // Constructor
  public Triunghi(string s, double w, double h) : base(w, h) {
    style = s;
  }

  // Construim un triunghi isoscel.
  public Triunghi(double x) : base(x) {
    style = "isoscel";
  }

  public double aria() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
  }
}

class Forme5 {
  public static void Main() {
    Triunghi t1 = new Triunghi();
    Triunghi t2 = new Triunghi("dreptunghic", 8.0, 12.0);
    Triunghi t3 = new Triunghi(4.0);
```

base poate fi utilizat la apelul diferitelor forme ale constructorului **Forma2D**.

```
    t1 = t2;

    Console.WriteLine("Informatii despre t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Aria triunghiului este " + t1.aria());

    Console.WriteLine();

    Console.WriteLine("Informatii despre t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Aria triunghiului este " + t2.aria());

    Console.WriteLine();

    Console.WriteLine("Informatii despre t3: ");
    t3.showStyle();
    t3.showDim();
    Console.WriteLine("Aria triunghiului este " + t3.aria());

    Console.WriteLine();
  }
}
```

Iată rezultatul afişat de această versiune:

```
Informatii despre t1:
Trunghiul este dreptunghic
Latimea si inaltimea sunt 8 si 12
Aria triunghiului este 48

Informatii despre t2:
Trunghiul este dreptunghic
Latimea si inaltimea sunt 8 si 12
Aria triunghiului este 48

Informatii despre t3:
Trunghiul este isoscel
Latimea si inaltimea sunt 4 si 4
Aria triunghiului este 8
```

Să recapitulăm conceptele esenţiale care stau în spatele lui **base**. Atunci când o clasă derivată conţine o clauză **base**, apelează constructorul clasei de bază din care derivă în mod direct. În concluzie, **base** se referă la clasa de bază situată imediat peste clasa apelantă. Acest fapt rămâne adevărat şi în ierarhiile cu mai multe niveluri. Puteţi transmite parametri constructorului clasei de bază specificându-i ca parametri ai lui **base**. Dacă nu există nici o clauză **base**, se apelează automat constructorul implicit al clasei de bază.

Exerciţii la minut

- Cum poate o clasă derivată să execute constructorul clasei sale de bază?
- Se pot transmite parametri prin intermediul lui **base**?
- Se referă **base** întotdeauna la constructorul clasei de bază situate imediat peste clasa apelantă?

Moştenirea şi ascunderea numelor

Este posibil ca o clasă derivată să definească un membru al cărui nume să coincidă cu numele unui membru al clasei de bază. În acest caz, membrul clasei de bază nu va mai fi vizibil în clasa derivată. Deşi din punct de vedere tehnic aceasta nu este o eroare în C#, compilatorul va genera un mesaj de avertisment. Acest avertisment vă informează că un nume a fost ascuns. Dacă aveţi cu adevărat intenţia de a ascunde un membru al clasei de bază, atunci puteţi evita acest avertisment precedând membrul clasei derivate cu cuvântul cheie **new**. Reţineţi însă că această utilizare a lui **new** nu are nimic în comun cu cea obişnuită, care creează instanţe ale claselor.

Mai jos, prezentăm un exemplu de ascundere a numelor:

```
// Un exemplu de ascundere a numelor cauzata de mostenire.
using System;

class A {
  public int i = 0;
}

// cream o clasa derivata.
class B : A {
  new int i;   // acest i il ascunde pe cel din clasa A

  public B(int b) {
    i = b;   // refera i-ul din clasa B
  }

  public void show() {
    Console.WriteLine("i din clasa derivata: " + i);
  }
}

class NameHiding {
  public static void Main() {
    B ob = new B(2);

    ob.show();
  }
}
```

> Membrul **i** din **A** este ascuns de **i**-ul din **B**. Remarcaţi utilizarea lui **new**.

- Prin specificarea unei clauze **base**.
- Da.
- Da.

Mai întâi, remarcaţi utilizarea lui **new**. Aici, acesta are rolul de a informa compilatorul că aţi creat în mod intenţionat o nouă variabilă **i**, care ascunde **i**-ul din clasa **A**. Fără acest **new**, compilatorul generează un avertisment.

Rezultatul afişat de acest program este prezentat mai jos:

```
i din clasa derivata: 2
```

Deoarece clasa **B** defineşte o variabilă instanţă proprie numită **i**, aceasta va ascunde **i**-ul din **A**. În concluzie, la invocarea metodei **show()** pentru un obiect de tipul **B**, se va afişa valoarea **i**-ului definit în clasa **B**, nu a celui din clasa **A**.

Utilizarea lui base pentru accesul la un nume ascuns

Există o a doua formă a lui **base** care se comportă oarecum similar cu **this**, referindu-se însă la clasa de bază a clasei derivate pentru care se utilizează. Această versiune are următoarea formă generală:

base.*membru*

Mai sus, *membru* poate desemna atât o metodă, cât şi o variabilă instanţă. Această formă a lui **base** se aplică îndeosebi în situaţiile în care numele unor membri din clasa derivată ascund membri cu acelaşi nume din clasa de bază. Să considerăm versiunea următoare a ierarhiei de clase din exemplul precedent:

```
// Utilizarea lui base pentru accesul la numele ascunse.
using System;

class A {
  public int i = 0;
}

// cream o clasa derivata.
class B : A {
  new int i;   // acest i il ascunde pe cel din clasa A

  public B(int a, int b) {
    base.i = a;   // descopera i-ul din A
    i = b; // refera i-ul din B
  }

  public void show() {
    // se afiseaza i-ul din A.
    Console.WriteLine("i din clasa de baza: " + base.i);

    // acum afisam i-ul din B.
    Console.WriteLine("i din clasa derivata: " + i);
  }
}

class UncoverName {
  public static void Main() {
    B ob = new B(1, 2);
```

base.i referă **i**-ul din clasa **A**.


```
    ob.show();
  }
}
```

Acest program afişează următorul text:

```
i din clasa de baza: 1
i din clasa derivata: 2
```

Chiar dacă variabila instanţă **i** din **B** ascunde **i**-ul din **A**, **base** permite accesul la variabila definită în clasa de bază.

Metodele ascunse pot fi de asemenea apelate prin intermediul lui **base**. Spre exemplu:

```
// Apelul unei metode ascunse.
using System;

class A {
  public int i = 0;

  // show() din A
  public void show() {
    Console.WriteLine("i din clasa de baza: " + i);
  }
}

// cream o clasa derivata.
class B : A {
  new int i; // acest i il ascunde pe cel din clasa A

  public B(int a, int b) {
    base.i = a;   // descopera i-ul din A
    i = b; // refera i-ul din B
  }

  // ascunde metoda show() din A.
  new public void show() {
    base.show(); // apel al metodei show() din A

    // acum afisam i-ul din B.
    Console.WriteLine("i din clasa derivata: " + i);
  }
}

class UncoverMethodName {
  public static void Main() {
    B ob = new B(1, 2);

    ob.show();
  }
}
```

Acest **show()** ascunde metoda din clasa **A**.

Apel al metodei ascunse **show()**.

Textul afişat de acest program este prezentat mai jos:

```
i din clasa de baza: 1
i din clasa derivata: 2
```

După cum se observă, **base.show()** apelează versiunea metodei **show()** din clasa de bază.

Un ultim amănunt: remarcați utilizarea lui **new** în program pentru a informa compilatorul că ați creat în mod intenționat o metodă numită **show()**, care ascunde metoda **show()** din clasa **A**.

Exerciții la minut

- Atunci când un nume din clasa de bază este ascuns în clasa derivată, ce cuvânt cheie trebuie să se găsească în fața acelui nume în clasa derivată?
- Pentru a descoperi un nume din clasa de bază, ce cuvânt cheie trebuie utilizat în clasa derivată?
- Poate apela o clasă derivată o metodă din clasa de bază al cărei nume a fost ascuns?

Proiectul 8-1: Extinderea clasei `Vehicle`

Pentru a demonstra puterea moștenirii, vom extinde clasa **Vehicle** pe care am dezvoltat-o în modulul 4. După cum vă amintiți, clasa **Vehicle** încapsulează informații despre vehicule, cum ar fi numărul de pasageri, capacitatea rezervorului de combustibil și rata de consum la 100 de km. Putem utiliza clasa **Vehicle** ca punct de pornire de la care să dezvoltăm clase mai specializate. De exemplu, camioanele sunt un tip aparte de vehicule. Un atribut important al oricărui camion îl constituie capacitatea sa de transport. În concluzie, pentru a crea clasa **Camion**, puteți moșteni **Vehicle**, la care adăugați o variabilă instanță pentru memorarea capacității de transport. În acest proiect, veți crea clasa **Camion**. Variabilele instanță din clasa **Vehicle** vor fi private, pentru accesul la valorile lor utilizându-se proprietăți.

Pas cu pas

1. Creați un fișier numit **CamionDemo.cs**, în care copiați ultima implementare a clasei **Vehicle** dn modulul 4.
2. Creați clasa **Camion** ca mai jos:

```
// Utilizam clasa Vehicle pentru a deriva clasa Camion.
class Camion : Vehicle {
  int pri_captran;   // capacitatea de transport, in kg.

  // constructorul clasei Camion.
  public Camion(int p, int ca, int cs, int ct) : base(p, ca, cs)
  {
    captran = ct;
  }

  // Proprietate utilizata pentru accesul la capacitatea de transport.
  public int captran {
    get { return pri_captran; }
```

- **new.**
- **base.**
- Da.


```
    set { pri_captran = value; }
  }
}
```

Clasa **Camion** definită mai sus moștenește clasa **Vehicle**, la care adaugă proprietatea **captran**. În concluzie, clasa **Camion** va conține toate atributele generale ale vehiculelor definite în clasa **Vehicle**. Trebuie adăugate numai elementele unice pentru această clasă.

3. Faceți ca variabilele instanță din clasa **Vehicle** să fie private și redenumiți-le după cum se observă mai jos.

```
int pri_pasageri;   // numarul de pasageri
int pri_capcomb;    // capacitatea rezervorului
int pri_cls;        // consumul de benzina la 100 km.
```

4. Adăugați proprietăți pentru accesul la aceste variabile, în modul următor:

```
// Proprietati.
public int pasageri {
  get { return pri_pasageri; }
  set { pri_pasageri = value; }
}

public int capcomb {
  get { return pri_capcomb; }
  set { pri_capcomb = value; }
}

public int cls {
  get { return pri_cls; }
  set { pri_cls = value; }
}
```

5. Iată listingul complet al programului care prezintă clasa **Camion**:

```
/*
  Proiectul 8-1

   Construirea unei clase derivate din Vehicle pentru camioane.
*/
using System;

class Vehicle {
  int pri_pasageri;   // nr. de pasageri
  int pri_capcomb;    // capacitatea rezervorului
  int pri_cls;        // consumul de combustibil, in litri la 100 km

  // Constructorul clasei Vehicle.
  public Vehicle(int p, int ca, int cs) {
    pasageri = p;
    capcomb = ca;
    cls = cs;
  }

  // Intoarce autonomia.
  public int aut() {
    return (100 * capcomb / cls);
  }

  // Calculeaza combustibilul necesar pe o distanta data.
```

```
  public double combnec(int dist) {
    return (double) cls * dist / 100.0;
  }

  // Proprietati.
  public int pasageri {
    get { return pri_pasageri; }
    set { pri_pasageri = value; }
  }

  public int capcomb {
    get { return pri_capcomb; }
    set { pri_capcomb = value; }
  }

  public int cls {
    get { return pri_cls; }
    set { pri_cls = value; }
  }
}

// Utilizam clasa Vehicle pentru a deriva clasa Camion.
class Camion : Vehicle {
  int pri_captran;  // capacitatea de transport, in kg.

  // constructorul clasei Camion.
  public Camion(int p, int ca, int cs, int ct)
  : base(p, ca, cs) {
    captran = ct;
  }

  // Proprietate utilizata pentru accesul la capacitatea de transport.
  public int captran {
    get { return pri_captran; }
    set ( pri_captran = value; }
  }
}

class CamionDemo {
  public static void Main() {
    // construim cateva instante
    Camion cam1 = new Camion(2, 1000, 40, 22000);
    Camion cam2 = new Camion(3, 150, 20, 1000);
    double consum;
    int dist = 407;

    consum = cam1.combnec(dist);

    Console.WriteLine("Cam1 poate transporta " + cam1.captran + " kilograme.");
    Console.WriteLine("Pe distanta de " + dist + " km, cam1
                        consuma " + consum + " litri de benzina.\n");

    consum = cam2.combnec(dist);

    Console.WriteLine("Cam2 poate transporta " + cam2.captran + " kilograme.");
```



```
    Console.WriteLine("Pe distanta de " + dist + " km, cam2
                        consuma " + consum + " litri de benzina.\n");
  }
}
```

6. Rezultatul afişat de acest program este prezentat mai jos:

```
Cam1 poate transporta 22000 kilograme.
Pe distanta de 407 km, cam1 consuma 162.8 litri de benzina.

Cam2 poate transporta 1000 kilograme.
Pe distanta de 407 km, cam2 consuma 81.4 litri de benzina.
```

7. Din clasa **Vehicle**, puteţi deriva multe alte clase. Spre exemplu, următoarea schemă creează clasa **VehTeren**, care memorează în plus garda la sol a vehiculului:

```
// Creeaza o clasa pt. reprezentarea vehiculelor de teren.
class VehTeren : Vehicle {
  int gardaSol; // garda la sol, in cm.

  // ...
}
```

Aspectul esenţial este că după ce aţi creat o clasă de bază care defineşte caracteristicile generale ale unui obiect, aceasta poate fi moştenită pentru a forma clase specializate. Fiecare clasă derivată adaugă numai atributele proprii. Aceasta este ideea fundamentală a moştenirii.

Crearea unei ierarhii pe mai multe niveluri

Până acum, am creat ierarhii simple de clase care conţin numai o clasă de bază şi o clasă derivată. Puteţi însă construi ierarhii care conţin oricâte niveluri de moştenire doriţi. După cum am menţionat, utilizarea unei clase derivate pe post de clasă de bază pentru o altă clasă este perfect corectă. De exemplu, dându-se trei clase numite **A**, **B** şi **C**, **C** poate deriva din **B**, care la rândul său poate deriva din **A**. În această situaţie, clasele derivate vor moşteni caracteristicile tuturor claselor de bază. În exemplul de mai sus, **C** va moşteni toate atributele claselor **B** şi **A**.

Pentru a studia utilitatea ierarhiilor multinivel, să considerăm următorul program. Acesta utilizează clasa derivată **Triunghi** pe post de clasă de bază pentru a crea clasa derivată numită **TriunghiColor**. **TriunghiColor** moşteneşte toate caracteristicile claselor **Triunghi** şi **Forma2D**, la care adaugă un câmp numit **culoare**, care memorează culoarea triunghiului.

```
// O ierarhie multinivel.
using System;

class Forma2D {
  double pri_width;   // privat
  double pri_height;  // privat

  // Constructorul implicit.
  public Forma2D() {
    width = height = 0.0;
```

```
  }

  // Constructorul clasei Forma2D.
  public Forma2D(double w, double h) {
    width = w;
    height = h;
  }

  // Construim un obiect cu latimea si inaltimea egale.
  public Forma2D(double x) {
    width = height = x;
  }

  // Proprietati pentru accesul la width si height.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width +
                      " si " + height);
  }
}

// clasa Triunghi este derivata din Forma2D.
class Triunghi : Forma2D {
  string style;   // privat

  /* Un constructor implicit. Acesta invoca automat
     constructorul implicit al clasei Forma2D. */
  public Triunghi() {
    style = "null";
  }

  // Constructor
  public Triunghi(string s, double w, double h) : base(w, h) {
    style = s;
  }

  // Construim un triunghi isoscel.
  public Triunghi(double x) : base(x) {
    style = "isoscel";
  }

  public double aria() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
  }
```



```
}

// Extindem clasa Triunghi.
class TriunghiColor : Triunghi {
  string culoare;

  public TriunghiColor(string c, string s,
         double w, double h) : base(s, w, h) {
    culoare = c;
  }

  // afiseaza culoarea.
  public void showColor() {
    Console.WriteLine("Culoarea este " + culoare);
  }
}

class Forme6 {
  public static void Main() {
    TriunghiColor t1 = new TriunghiColor("Albastru", "dreptunghic", 8.0, 12.0);
    TriunghiColor t2 = new TriunghiColor("Rosu", "dreptunghic", 2.0, 2.0);

    Console.WriteLine("Informatii despre t1: ");
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    Console.WriteLine("Aria triunghiului este " + t1.aria());

    Console.WriteLine();

    Console.WriteLine("Informatii despre t2: ");
    t2.showStyle();
    t2.showDim();
    t2.showColor();
    Console.WriteLine("Aria triunghiului este " + t2.aria());
  }
}
```

TriunghiColor moşteneşte **Triunghi**, care derivă din **Forma2D**, deci **TriunghiColor** include toţi membrii claselor **Triunghi** şi **Forma2D**.

Un obiect **TriunghiColor** poate apela metode definite în propria clasă sau în clasele de bază.

Rezultatul afişat de acest program este prezentat mai jos:

```
Informatii despre t1:
Triunghiul este dreptunghic
Latimea si lungimea sunt 8 si 12
Culoarea este Albastru
Aria triunghiului este 48

Informatii despre t2:
Triunghiul este isoscel
Latimea si lungimea sunt 2 si 2
Culoarea este Rosu
Aria triunghiului este 2
```

Datorită moştenirii, clasa **TriunghiColor** poate beneficia de clasele **Triunghi** şi **Forma2D** definite anterior, la care adaugă numai informaţia suplimentară necesară aplicaţiei particulare pentru care a fost proiectată. Acesta este unul din motivele importante ce conferă valoare moştenirii – posibilitatea reutilizării codului.

Exemplul demonstrează încă un aspect important: **base** referă întotdeauna constructorul din clasa de bază cea mai apropiată. Apelul **base** din clasa **TriunghiColor** se referă la constructorul din clasa **Triunghi**. În clasa **Triunghi**, **base** apelează constructorul clasei **Forma2D**. Dacă într-o ierarhie de clase constructorul unei clase de bază necesită parametri, toate clasele derivate trebuie să transmită acei parametri „mai departe". Aceasta rămâne adevărat indiferent dacă o clasă derivată necesită sau nu parametri proprii.

Când se apelează constructorii?

În discuția anterioară despre moștenire și ierarhiile de clase, este posibil să vă fi gândit deja la o întrebare importantă: la crearea unei instanțe a unei clase derivate, care constructor se execută mai întâi? Cel al clasei derivate sau cel definit de către clasa de bază? De exemplu, fie **B** o clasă care este derivată din clasa **A**, atunci se apelează mai întâi constructorul lui **A** sau constructorul lui **B**? Răspunsul este că, într-o ierarhie de clase, constructorii sunt apelați în ordinea derivării, de la clasa de bază spre clasele derivate. Mai mult, această ordine se respectă indiferent dacă se apelează sau nu **base**. Dacă nu se utilizează **base**, atunci se va executa constructorul implicit (fără parametri) al fiecărei clase de bază. Programul următor prezintă momentul execuției constructorilor:

```
// Prezinta momentul executiei constructorilor.
using System;

// cream o clasa de baza.
class A {
  public A() {
    Console.WriteLine("Constructorul clasei A.");
  }
}

// cream o clasa derivata din A.
class B : A {
  public B() {
    Console.WriteLine("Constructorul clasei B.");
  }
}

// cream o clasa derivata din B.
class C : B {
  public C() {
    Console.WriteLine("Constructorul clasei C.");
  }
}

// class OrdineaConstructori {
  public static void Main() {
    C c = new C();
  }
}
```


Rezultatul acestui program este prezentat mai jos:

```
Constructorul clasei A.
Constructorul clasei B.
Constructorul clasei C.
```

După cum se observă, constructorii au fost apelați în ordinea derivării.

Execuția constructorilor în ordinea derivării este logică. Deoarece o clasă de bază nu își cunoaște clasele derivate, inițalizările efectuate de aceasta trebuie să fie separate și pe cât posibil anterioare față de inițializările efectuate de către clasa derivată. În concluzie, constructorul clasei de bază trebuie să se execute primul.

Atribuirea unor instanțe ale claselor derivate variabilelor de tipul clasei de bază

După cum știți deja, limbajul C# este puternic tipizat. În afara conversiilor standard și a promovărilor automate care se aplică asupra tipurilor simple, compatibilitatea tipurilor este strict controlată. În consecință, o variabilă referință de tipul unei anumite clase nu va putea, în majoritatea cazurilor, referi un obiect aparținând unei alte clase. De exemplu, să considerăm următorul program:

```
// Nu se va putea compila acest program.
class X {
  int a;

  public X(int i) { a = i; }
}

class Y {
  int a;

  public Y(int i) { a = i; }
}

class ReferinteIncompatibile {
  public static void Main() {
    X x = new X(10);
    X x2;
    Y y = new Y(5);

    x2 = x;   // corect, referinte de acelasi tip

    x2 = y;   // eroare, referinte de tipuri diferite
  }
}
```

Aceste referințe nu sunt compatibile.

Mai sus, deși clasele **X** și **Y** sunt practic identice, nu este posibil să atribuim o referință **X** unui obiect de tipul **Y**, deoarece acestea au tipuri diferite. În general, variabilele referință nu pot referi decât obiecte de același tip cu ele.

Regula de compatibilitate strictă a tipurilor în C# admite, totuși, o excepție importantă. Unei variabile referință de tipul unei clase de bază i ...

referință către un obiect aparținând oricărei clase derivate din clasa de bază. Iată un exemplu:

```
// Referinta catre o clasa de baza poate referi un obiect
// apartinand unei clase derivate.
using System;

class X {
  public int a;

  public X(int i) { a = i; }
}

class Y : X {
  public int b;

  public Y(int i, int j) : base(j) {
    b = i;
  }
}

class RefBaza {
  public static void Main() {
    X x = new X(10);
    X x2;
    Y y = new Y(5, 6);

    x2 = x; // corect, fiind de acelasi tip
    Console.WriteLine("x2.a: " + x2.a);

    x2 = y; // corect, deoarece Y deriva din X
    Console.WriteLine("x2.a: " + x2.a);

    // referintele catre X cunosc numai membrii clasei de baza X
    x2.a = 19; // corect
//    x2.b = 27; // Eroare, clasa X nu are membrul b
  }
}
```

Este corect deoarece Y este derivat din X și astfel x2 se poate referi la y.

Clasa `Y` derivă acum din `X`; este deci permis ca lui `x2` să i se atribuie o referință către un obiect `Y`.

Este foarte important să înțelegeți că tipul variabilei referință – și nu tipul obiectului pe care aceasta îl referă – determină membrii care pot fi accesați. În consecință, atunci când atribuiți o referință către o instanță a clasei derivate unei variabile referință de tipul clasei de bază, veți avea acces numai la părțile obiectului definite în clasa de bază. Din această cauză, `x2` nu are acces la câmpul `b`, chiar atunci când referă un obiect `Y`. Aceasta este normal, deoarece clasa de bază nu știe ce membri a mai adăugat clasa derivată. Din acest motiv, am comentat intenționat ultima linie din program.

Chiar dacă discuția anterioară pare puțin ezoterică, are câteva aplicații practice importante. Prima dintre acestea este prezentată în continuare. Cealaltă va fi descrisă ulterior în acest modul, după prezentarea metodelor virtuale.

Un caz important în care se atribuie referințe către clasele derivate unor variabile de tipul clasei de bază este la apelul constructorilor într-o ierarhie de clase. După cum știți, clasele definesc în mod uzual constructori care acceptă ca parametru o instanță a lor. Aceasta permite fiecărei clase să construiască o copie a unui obiect. Clasele derivate dintr-o astfel de clasă pot beneficia la rândul lor de această facilitate. De exemplu, să considerăm următoarele versiuni ale claselor `Forma2D` și `Triunghi`. Ambele adaugă constructori care primesc ca parametru un obiect.

```
// Atribuirea unei referinte catre clasa derivata unei variabile
// de tipul clasei de baza.

using System;

class Forma2D {
  double pri_width;  // privat
  double pri_height; // privat

  // Constructorul implicit.
  public Forma2D() {
    width = height = 0.0;
  }

  // Constructorul clasei Forma2D.
  public Forma2D(double w, double h) {
    width = w;
    height = h;
  }

  // Construim un obiect cu latimea si inaltimea egale.
  public Forma2D(double x) {
    width = height = x;
  }

  // Constructor care copiaza un obiect existent.
  public Forma2D(Forma2D ob) {
    width = ob.width;
    height = ob.height;
  }

  // Proprietati pentru accesul la width si height.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width +
                      " si " + height);
  }
}
```

Construiește un obiect copiind un altul.

```
// clasa Triunghi este derivata din Forma2D.
class Triunghi : Forma2D {
  string style;    // privat

  // Un constructor implicit.
  public Triunghi() {
    style = "null";
  }

  // Constructorul clasei Triunghi
  public Triunghi(string s, double w, double h) : base(w, h) {
    style = s;
  }

  // Construim un triunghi isoscel.
  public Triunghi(double x) : base(x) {
    style = "isoscel";
  }

  // Constructor de copiere.
  public Triunghi(Triunghi ob) : base(ob) {
    style = ob.style;
  }

  public double aria() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
  }
}

class Forme7 {
  public static void Main() {
    Triunghi t1 = new Triunghi("dreptunghic", 8.0, 12.0);

    // copiem obiectul t1
    Triunghi t2 = new Triunghi(t1);

    Console.WriteLine("Informatii despre t1: ");
    t1.showStyle();
    t1.showDim();
    Console.WriteLine("Aria triunghiului este " + t1.aria());

    Console.WriteLine();

    Console.WriteLine("Informatii despre t2: ");
    t2.showStyle();
    t2.showDim();
    Console.WriteLine("Aria triunghiului este " + t2.aria());
  }
}
```

Transmite o referință **Triunghi** constructorului clasei **Forma2D**.

În acest program, **t2** este construit prin copierea lui **t1**, fiind deci identic cu acesta. Rezultatul programului este prezentat mai jos:


```
Informatii despre t1:
Triunghiul este dreptunghic
Latimea si inaltimea sunt 8 si 12
Aria triunghiului este 48

Informatii despre t2:
Triunghiul este dreptunghic
Latimea si inaltimea sunt 8 si 12
Aria triunghiului este 48
```

Remarcați îndeosebi următorul constructor al clasei **Triunghi**:

```
// Constructor de copiere.
public Triunghi(Triunghi ob) : base(ob) {
  style = ob.style;
}
```

Acesta primește ca parametru un obiect de tipul **Triunghi**, pe care îl transmite (prin intermediul lui **base**) următorului constructor al clasei **Forma2D**:

```
// Constructor care copiaza un obiect existent.
public Forma2D(Forma2D ob) {
  width = ob.width;
  height = ob.height;
}
```

Constructorul clasei **Forma2D()** așteaptă un obiect **Forma2D**, însă **Triunghi()** îi transmite un obiect de tipul **Triunghi**. După cum am explicat, motivul pentru care se poate proceda astfel este că o referință de tipul clasei de bază poate referi o instanță a unei clase derivate. Este deci perfect corect ca **Forma2D()** să primească o referință către un obiect aparținând unei clase derivate din **Forma2D**. Deoarece constructorul **Forma2D()** inițializează numai acele entități din instanța clasei derivate care sunt membre ale clasei **Forma2D**, nu are importanță dacă obiectul mai conține și alți membri adăugați de clasele derivate.

Exerciții la minut

- Poate o clasă derivată să fie clasa de bază a unei alte clase?
- Într-o ierarhie de clase, care este ordinea în care se apelează constructorii?
- Știind că **Jet** este o clasă derivată din **Airplane**, este posibil ca o referință **Airplane** să refere un obiect de tipul **Jet**?

- Da.
- Constructorii sunt apelați în ordinea stabilită de derivare.
- Da. În orice situație, o referință de tipul clasei de bază poate referi o instanță a unei clase derivate. Reciproca nu este adevărată.

Metode virtuale. Extinderea metodelor

O *metodă virtuală* este o metodă declarată ca **virtual** într-o clasă de bază şi redefinită apoi în cel puţin una dintre clasele derivate. Fiecare dintre clasele derivate poate deci dispune de o versiune proprie a unei metode virtuale. Metodele virtuale sunt interesante atunci când sunt apelate prin intermediul unei referinţe de tipul clasei de bază. În această situaţie, compilatorul de C# determină versiunea care va fi apelată ţinând cont de *tipul obiectului referit* de referinţă – determinarea fiind efectuată *la momentul execuţiei*. În consecinţă, pentru obiecte referite de tipuri diferite, se execută versiuni diferite ale metodei virtuale. Cu alte cuvinte, tipul obiectului care este referit (şi nu tipul referinţei) este cel care stabileşte ce versiune a metodei virtuale se va executa. Ca urmare, dacă o clasă de bază conţine o metodă virtuală şi există clase derivate din clasa de bază, atunci când se referă tipuri diferite de obiecte prin intermediul unei referinţe de tipul clasei de bază, se vor executa versiuni diferite ale metodei virtuale.

Declararea unei metode virtuale în corpul unei clase de bază se face precedând declaraţia metodei cu cuvântul cheie **virtual**. La redefinirea metodei într-o clasă derivată, se utilizează modificatorul **override**. Procesul de redefinire a unei metode virtuale în corpul unei clase derivate poartă numele de *extindere a metodei*. La extinderea unei metode, numele şi signatura de tipuri a metodei extinse trebuie să coincidă cu numele şi signatura metodei virtuale care se extinde. Reţineţi de asemenea că metodele virtuale nu pot fi declarate ca **static** sau **abstract** (modificator care va fi prezentat ulterior în acest modul).

Extinderea metodelor reprezintă baza pentru unul din cele mai puternice concepte implementate în C#: *apelarea dinamică a metodelor*. Apelarea dinamică a metodelor este mecanismul prin care rezolvarea apelului unei funcţii extinse este amânată de la momentul compilării până la momentul execuţiei programului. Apelarea dinamică a metodelor este extrem de importantă, deoarece este modalitatea prin care limbajul C# implementează polimorfismul la execuţie.

În continuare prezentăm un exemplu care demonstrează utilizarea metodelor virtuale şi conceptul de extindere:

```
// Prezinta utilizarea metodelor virtuale.
using System;

class Baza {
  // Crearea unei metode virtuale in clasa de baza.
  public virtual void cine() {
    Console.WriteLine("metoda cine() din clasa Baza");
  }
}
```

Declararea unei metode virtuale.


```
class Derivata1 : Baza {
  // Extindem metoda cine() intr-o clasa derivata.
  public override void cine() {
    Console.WriteLine("metoda cine() din clasa Derivata1");
  }
}

class Derivata2 : Baza {
  // Extindem metoda cine() in alta clasa derivata.
  public override void cine() {
    Console.WriteLine("metoda cine() din clasa Derivata2");
  }
}

class ExtDemo {
  public static void Main() {
    Baza baseOb = new Baza();
    Derivata1 dOb1 = new Derivata1();
    Derivata2 dOb2 = new Derivata2();

    Baza baseRef; // referinta de tipul clasei de baza

    baseRef = baseOb;
    baseRef.cine();

    baseRef = dOb1;
    baseRef.cine();

    baseRef = dOb2;
    baseRef.cine();
  }
}
```

Extinderea metodei virtuale.

În fiecare caz, versiunea metodei **cine()** care este apelată se determină la execuţie în funcţie de tipul obiectului referit.

Rezultatul afişat de acest program este prezentat mai jos:

```
metoda cine() din clasa Baza
metoda cine() din clasa Derivata1
metoda cine() din clasa Derivata2
```

Programul creează o clasă de bază numită **Baza**, din care derivează alte două clase, numite **Derivata1** şi **Derivata2**. Clasa **Baza** defineşte o metodă virtuală numită **cine()**, care este extinsă în clasele derivate. În corpul metodei **Main()**, se declară obiecte de tipurile **Baza**, **Derivata1** şi **Derivata2**. Tot aici se declară şi o referinţă de tipul **Baza**, numită **baseRef**. Programul atribuie apoi lui **baseRef** o referinţă către fiecare tip de obiecte, utilizând apoi referinţa pentru apelul metodei **cine()**. După cum arată şi rezultatul, versiunea metodei **cine()** care se execută este determinată ţinând cont de tipul obiectului referit la momentul apelului şi nu de tipul referinţei **baseRef**.

Nu este obligatoriu ca o metodă virtuală să fie extinsă. Dacă o clasă derivată nu conţine o versiune proprie a unei metode virtuale, atunci se utilizează versiunea din clasa de bază. De exemplu:

```
/* Atunci cand o metoda virtuala nu este extinsa, se utilizeaza
   versiunea din clasa de baza. */
using System;
```

```
class Baza {
  // Crearea unei metode virtuale in clasa de baza.
  public virtual void cine() {
    Console.WriteLine("metoda cine() din clasa Baza");
  }
}

class Derivata1 : Baza {
  // Extindem metoda cine() intr-o clasa derivata.
  public override void cine() {
    Console.WriteLine("metoda cine() din clasa Derivata1");
  }
}

class Derivata2 : Baza {   <-- Metoda cine() nu este extinsă aici.
  // Aceasta clasa nu mai extinde metoda cine().
}

class NoExtDemo {
  public static void Main() {
    Baza baseOb = new Baza();
    Derivata1 dOb1 = new Derivata1();
    Derivata2 dOb2 = new Derivata2();

    Baza baseRef; // referinta de tipul clasei de baza

    baseRef = baseOb;
    baseRef.cine();

    baseRef = dOb1;
    baseRef.cine();

    baseRef = dOb2;
    baseRef.cine(); // apeleaza cine() din clasa Baza   <-- Apelul metodei cine() din Baza.
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
metoda cine() din clasa Baza
metoda cine() din clasa Derivata1
metoda cine() din clasa Baza
```

Clasa **Derivata2** nu mai extinde metoda **cine()**. Ca urmare, atunci când se apelează **cine()** pentru un obiect de tipul **Derivata2**, se execută metoda **cine()** din clasa **Baza**.

Ce avantaje ne aduce extinderea metodelor?

Extinderea metodelor permite limbajului C# să implementeze conceptul de polimorfism la execuție. Polimorfismul este un concept fundamental în programarea orientată spre obiecte din următorul motiv: permite unei clase generale să specifice metode comune tuturor claselor derivate din ea, clasele derivate definind implementări particulare ale unora sau tuturor acestor metode. Extinderea metodelor reprezintă o altă modalitate în care limbajul C# implementează aspectul „o singură interfață, mai multe metode" al polimorfismului.

O parte a rețetei de utilizare cu succes a polimorfismului o constituie înțelegerea faptului că mulțimea claselor de bază și claselor derivate formează o ierarhie ordonată crescător din punct de vedere al specializării. Dacă este definită corect, clasa de bază pune la dispoziție toate elementele pe care clasele derivate le pot utiliza direct. Tot clasa de bază definește și metodele pe care clasa derivată trebuie să le implementeze în mod independent. Aceasta pe de o parte oferă clasei derivate suficientă flexibilitate în definirea propriilor metode, iar pe de altă parte impune utilizarea unei interfețe consistente. Prin combinarea moștenirii cu extinderea metodelor, o clasă de bază poate deci defini forma generală a metodelor care vor fi utilizate de toate clasele sale derivate.

Sfatul expertului

Întrebare: Pot fi proprietățile virtuale?

Răspuns: Da. Proprietățile pot fi la rândul lor modificate cu ajutorul cuvântului cheie **virtual** și extinse utilizând **override**. Același lucru este valabil și pentru indexări.

O aplicație a metodelor virtuale

Pentru a înțelege mai bine puterea metodelor virtuale, o vom aplica în cazul concret al clasei **Forma2D**. În exemplele anterioare, fiecare din clasele derivate din **Forma2D** definește o metodă numită **aria()**. Aceasta ne conduce la ideea că ar fi mai potrivit să declarăm metoda **aria()** ca virtuală în clasa **Forma2D**, permițând ulterior fiecărei clase derivate să o extindă, reflectând modalitatea de calcul al ariei pentru tipul particular de formă pe care clasa derivată o încapsulează. Aceasta se realizează în programul următor. Pentru simplitate, programul adaugă la clasa de bază **Forma2D** un câmp care reprezintă numele formei. (În acest mod, este mult mai simplu să distingem clasele derivate între ele.)

```
// Utilizarea metodelor virtuale si polimorfismului.
using System;

class Forma2D {
  double pri_width;   // privat
  double pri_height;  // privat
  string pri_name;  // privat

  // Constructorul implicit.
  public Forma2D() {
    width = height = 0.0;
    name = "null";
  }

  // Constructorul cu parametri al clasei Forma2D.
  public Forma2D(double w, double h, string n) {
```

```
    width = w;
    height = h;
    name = n;
  }

  // Construim un obiect cu latimea si inaltimea egale.
  public Forma2D(double x, string n) {
    width = height = x;
    name = n;
  }

  // Constructor care copiaza un obiect existent.
  public Forma2D(Forma2D ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
  }

  // Proprietati pentru accesul la width, height si name.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public string name {
    get { return pri_name; }
    set { pri_name = value; }
  }

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width + " si " + height);
  }

  public virtual double aria() {
    Console.WriteLine("Metoda aria() trebuie extinsa");
    return 0.0;
  }
}

// O clasa derivata din Forma2D care reprezinta triunghiuri.
class Triunghi : Forma2D {
  string style;   // privat

  // Un constructor implicit.
  public Triunghi() {
    style = "null";
  }

  // Constructorul clasei Triunghi
  public Triunghi(string s, double w, double h) : base(w, h, "triunghi") {
    style = s;
  }
```

Metoda **aria()** definită în clasa **Forma2D** este acum virtuală.


```
  // Construim un triunghi isoscel.
  public Triunghi(double x) : base(x, "triunghi") {
    style = "isoscel";
  }

  // Constructor de copiere.
  public Triunghi(Triunghi ob) : base(ob) {
    style = ob.style;
  }

  // Extindem metoda aria() pentru clasa Triunghi.
  public override double aria() {
    return width * height / 2;
  }

  public void showStyle() {
    Console.WriteLine("Triunghiul este " + style);
  }
}

// O clasa derivata din Forma2D care reprezinta dreptunghiuri.
class Dreptunghi : Forma2D {
  // Constructorul clasei Dreptunghi
  public Dreptunghi(double w, double h) :
    base(w, h, "dreptunghi") { }

  // Construim un patrat.
  public Dreptunghi(double x) : base(x, "dreptunghi") { }

  // Constructor de copiere.
  public Dreptunghi(Dreptunghi ob) : base(ob) { }

  public bool estePatrat() {
    if(width == height) return true;
    return false;
  }

  // Extindem metoda aria() pentru clasa Dreptunghi.
  public override double aria() {
    return width * height;
  }
}

class FormeDinamice {
  public static void Main() {
    Forma2D [] forme = new Forma2D[5];

    forme[0] = new Triunghi("dreptunghic", 8.0, 12.0);
    forme[1] = new Dreptunghi(10);
    forme[2] = new Dreptunghi(10, 4);
    forme[3] = new Triunghi(7.0);
    forme[4] = new Forma2D(10, 20, "generic");

    for(int i = 0; i < forme.Length; i++) {
      Console.WriteLine("Obiectul este " + forme[i].name);
      Console.WriteLine("Aria este " + forme[i].aria());
```

Extindem **aria()** pentru clasa **Triunghi**.

Extindem **aria()** pentru clasa **Dreptunghi**.

Pentru fiecare formă, se apelează versiunea corespunzătoare a metodei **aria()**.

```
        Console.WriteLine();
    }
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Obiectul este triunghi
Aria este 48

Obiectul este dreptunghi
Aria este 100

Obiectul este dreptunghi
Aria este 40

Obiectul este triunghi
Aria este 24.5

Obiectul este generic
Metoda aria() trebuie extinsa
Aria este 0
```

Să studiem acest program mai amănunţit. Mai întâi, după cum am explicat, metoda `aria()` este declarată `virtual` în clasa `Forma2D`, fiind apoi extinsă în clasele `Triunghi` şi `Dreptunghi`. În clasa `Forma2D`, metoda `aria()` are o implementare formală, care doar informează utilizatorul că metoda trebuie extinsă în clasele derivate. Versiunile extinse ale metodei `aria()` conţin implementări potrivite tipurilor de obiecte încapsulate de clasele derivate. Dacă trebuie, de exemplu, să implementaţi o clasă care încapsulează elipse, metoda `aria()` va calcula aria unei elipse.

Există şi un alt aspect care merită remarcat în programul anterior. Observaţi că în metoda `Main()`, tabloul `forme` este declarat ca un tablou de obiecte `Forma2D`. Elementelor acestui tablou le sunt însă atribuite referinţe de tipurile `Triunghi`, `Dreptunghi` şi `Forma2D`. Aceasta este corect, deoarece o referinţă având ca tip o clasă de bază poate referi o instanţă a unei clase derivate. Programul parcurge iterativ tabloul, afişând informaţiile despre toate obiectele pe care acesta le conţine. Deşi este relativ simplu, acest program demonstrează foarte bine puterea mecanismelor de moştenire şi de extindere a metodelor. Tipul obiectului referit de către o referinţă de tipul clasei de bază este determinat la execuţie şi utilizat în consecinţă. Pentru orice obiect derivat din clasa `Forma2D`, aria se poate determina apelând metoda `aria()`. Interfaţa acestei operaţii este aceeaşi, indiferent de tipul formei încapsulate.

Exerciţii la minut

- Ce este o metodă virtuală? Cum se poate extinde o astfel de metodă?
- În ce constă importanţa metodelor virtuale?
- Ce versiune a unei metode se execută atunci când metoda este apelată prin intermediul unei referinţe de tipul clasei de bază?

Utilizarea claselor abstracte

În anumite situaţii, este necesar să creaţi o clasă de bază care stabileşte numai un şablon general comun pentru toate clasele derivate, fiecare dintre acestea completând detaliile caracteristice. O astfel de clasă fixează natura metodelor pe care clasele derivate trebuie să le implementeze, fără însă a oferi ea însăşi o implementare a acestor metode. O posibilitate în care această situaţie apare efectiv este când o clasă de bază nu este capabilă să creeze o implementare semnificativă pentru o metodă. Aşa stau lucrurile şi cu clasa `Forma2D` utilizată în exemplul anterior. Definiţia metodei `aria()` este pur şi simplu formală. Aceasta nu va calcula şi nu va afişa aria nici unui tip de obiecte.

Atunci când veţi crea propriile dumneavoastră biblioteci de clase, veţi întâlni destul de frecvent metode care nu admit nici o definiţie semnificativă în contextul claselor lor de bază. Puteţi rezolva această problemă în două moduri. O soluţie, prezentată în exemplul anterior, este de a afişa pur şi simplu un mesaj de avertisment. Această soluţie se poate dovedi utilă în anumite situaţii – ca de exemplu la depanare – dar de regulă nu este adecvată. Pot exista metode care *trebuie neapărat* extinse în clasa derivată pentru a căpăta o semnificaţie. Să considerăm de exemplu clasa `Triunghi`. Dacă metoda `aria()` nu este definită, implementarea din clasa de bază nu are nici un sens. În acest caz, este necesară o modalitate care să ne asigure că în clasa derivată toate metodele necesare sunt într-adevăr extinse. Soluţia oferită de limbajul C# la această problemă este utilizarea metodelor abstracte.

O metodă abstractă se poate crea specificând modificatorul de tip `abstract`. Metodele abstracte nu au corp şi, în concluzie, nu sunt implementate în cadrul clasei de bază. Clasele derivate trebuie deci să le extindă – nu se admite să utilizeze versiunea definită în clasa de bază. După cum intuiţi deja, metodele abstracte sunt în mod implicit virtuale; în consecinţă, nu este cazul să adăugaţi şi modificatorul `virtual`. Utilizarea ambilor modificatori, `virtual` şi `abstract`, împreună constituie de fapt o eroare.

- O metodă virtuală se declară ca `virtual` într-o clasă de bază, fiind apoi extinsă în clasele derivate. Extinderea metodelor virtuale se realizează utilizând modificatorul `override`.
- Metodele virtuale constituie una din modalităţile în care limbajul C# implementează conceptul de polimorfism.
- Versiunea unei metode extinse care se va executa este determinată de tipul obiectului referit la momentul apelului. Această determinare se efectuează, deci, la momentul execuţiei.

Pentru a declara o metodă abstractă, utilizați următoarea formă generală:

abstract *tip nume(lista-parametri)*;

După cum se observă, metoda nu are corp. Modificatorul **abstract** poate fi utilizat numai pentru metodele obişnuite. Nu este permisă utilizarea sa în cazul metodelor **static**.

O clasă care conține cel puțin o metodă abstractă trebuie de asemenea să fie declarată ca abstractă, precedând declarația **class** cu specificatorul **abstract**. Deoarece clasele abstracte nu definesc implementări complete, aceste clase nu se pot instanția. Încercarea de a crea o instanță a unei clase abstracte utilizând **new** va produce o eroare la compilare.

Atunci când o clasă derivată moşteneşte caracteristicile unei clase abstracte, trebuie să implementeze toate metodele abstracte din clasa de bază. În caz contrar, şi clasa derivată trebuie specificată ca **abstract**. Atributul **abstract** se moşteneşte până în momentul în care se realizează o implementare completă.

Puteți îmbunătăți performanțele clasei **Forma2D** din programul cu obiecte bidimensionale, declarând clasa ca abstractă. Întrucât conceptul de arie nu are nici o semnificație pentru figuri bidimensionale nedefinite, versiunea următoare a programului anterior va declara metoda **aria()** ca **abstract** în clasa **Forma2D**, aceasta devenind la rândul său abstractă. Aceasta, desigur, înseamnă că toate clasele derivate din **Forma2D** trebuie să extindă metoda **aria()**.

```
// Crearea unei clase abstracte.
using System;

abstract class Forma2D {
  double pri_width;   // privat
  double pri_height;  // privat
  string pri_name;    // privat

  // Constructorul implicit.
  public Forma2D() {
    width = height = 0.0;
    name = "null";
  }

  // Constructorul cu parametri al clasei Forma2D.
  public Forma2D(double w, double h, string n) {
    width = w;
    height = h;
    name = n;
  }

  // Construim un obiect cu latimea si inaltimea egale.
  public Forma2D(double x, string n) {
    width = height = x;
    name = n;
  }

  // Constructor care copiaza un obiect existent.
```

Clasa **Forma2D** este acum abstractă.


```
  public Forma2D(Forma2D ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
  }

  // Proprietati pentru accesul la width, height si name.
  public double width {
    get { return pri_width; }
    set { pri_width = value; }
  }

  public double height {
    get { return pri_height; }
    set { pri_height = value; }
  }

  public string name {
    get { return pri_name; }
    set { pri_name = value; }
  }

  public void showDim() {
    Console.WriteLine("Lungimea si latimea sunt " + width +
                      " si " + height);
  }

  // metoda aria() este acum abstracta.
  public abstract double aria();
}

// O clasa derivata din Forma2D care reprezinta triunghiuri.
class Triunghi : Forma2D {
  string style; // privat

  // Un constructor implicit.
  public Triunghi() {
    style = "null";
  }

  // Constructorul clasei Triunghi
  public Triunghi(string s, double w, double h) :
  base(w, h, "triunghi") {
    style = s;
  }

  // Construim un triunghi isoscel.
  public Triunghi(double x) : base(x, "triunghi") {
    style = "isoscel";
  }

  // Constructor de copiere.
  public Triunghi(Triunghi ob) : base(ob) {
    style = ob.style;
  }
```

Metoda **aria()** definită de clasa **Forma2D** este acum abstractă.

```
    // Extindem metoda aria() pentru clasa Triunghi.
    public override double aria() {
      return width * height / 2;
    }

    public void showStyle() {
      Console.WriteLine("Triunghiul este " + style);
    }
}

// O clasa derivata din Forma2D care reprezinta dreptunghiuri.
class Dreptunghi : Forma2D {
  // Constructorul clasei Dreptunghi
  public Dreptunghi(double w, double h) : base(w, h, "dreptunghi") { }

  // Construim un patrat.
  public Dreptunghi(double x) : base(x, "dreptunghi") { }

  // Constructor de copiere.
  public Dreptunghi(Dreptunghi ob) : base(ob) { }

  public bool estePatrat() {
    if(width == height) return true;
    return false;
  }

  // Extindem metoda aria() pentru clasa Dreptunghi.
  public override double aria() {
    return width * height;
  }
}

class FormeAbs {
  public static void Main() {
    Forma2D [] forme = new Forma2D[4];

    forme[0] = new Triunghi("dreptunghic", 8.0, 12.0);
    forme[1] = new Dreptunghi(10);
    forme[2] = new Dreptunghi(10, 4);
    forme[3] = new Triunghi(7.0);

    for(int i = 0; i < forme.Length; i++) {
      Console.WriteLine("Obiectul este " + forme[i].name);
      Console.WriteLine("Aria este " + forme[i].aria());

      Console.WriteLine();
    }
  }
}
```

Extindem metoda **aria()** în clasa **Triunghi**.

Extindem metoda **aria()** în clasa **Dreptunghi**.

După cum arată acest program, toate clasele derivate din **Forma2D** trebuie să extindă metoda **aria()** (altfel vor fi declarate **abstract**). Pentru a vă convinge de acest adevăr, încercați să creați o clasă derivată care nu extinde metoda **aria()**. Se va produce o eroare la momentul compilării. Desigur, este încă posibil să creați referințe către obiecte de tipul **Forma2D**, iar programul chiar face aceasta. Nu mai este însă posibil să

declarați obiecte de tipul **Forma2D**. Din acest motiv, în metoda **Main()**, tabloul **forme** are aici numai 4 elemente, crearea unui obiect generic **Forma2D** nemaifiind permisă.

Un ultim amănunt: remarcați că metoda **Forma2D()** conține încă metoda **showDim()**, care nu este influențată de **abstract**. Este perfect corect – și totodată destul de frecvent – ca o clasă abstractă să conțină metode care nu sunt abstracte, pe care clasele derivate le pot utiliza nemodificate. Numai metodele declarate ca **abstract** trebuie extinse în cadrul claselor derivate.

Exerciții la minut

- Ce este o metodă abstractă? Cum se poate crea o astfel de metodă?
- Ce este o clasă abstractă?
- Se pot crea instanțe ale claselor abstracte?

Utilizarea lui sealed pentru a împiedica moștenirea

Oricât de puternică și de utilă ar fi moștenirea, uneori este necesar să o împiedicați. Spre exemplu, puteți avea o clasă care încapsulează secvența de inițializare a unui dispozitiv fizic specializat, cum ar fi un monitor medical. În acest caz, nu trebuie să lăsați utilizatorilor clasei dumneavoastră posibilitatea de a modifica inițializarea monitorului, pe care o vor efectua foarte probabil într-un mod incorect. Indiferent de motiv, în C# este foarte simplu să nu permiteți ca o clasă să fie moștenită, utilizând cuvântul cheie **sealed**.

Pentru a împiedica o clasă să fie moștenită, trebuie să precedați declarația clasei cu cuvântul cheie **sealed**. După cum intuiți, este incorect să declarați o clasă simultan **abstract** și **sealed**, deoarece o clasă abstractă este incompletă, lăsând în sarcina claselor derivate specificarea unor implementări complete.

Mai jos prezentăm un exemplu de clasă declarată **sealed**:

```
sealed class A {
  // ...
}

// Urmatoarea declaratie este incorecta.
class B : A { // EROARE !!! Clasa A nu poate fi derivata.
  // ...
}
```

Această clasă nu poate fi moștenită.

- O metodă abstractă este o metodă fără corp. Aceasta constă deci numai din tipul rezultatului, nume și lista de parametri și este precedată de cuvântul cheie **abstract**.
- Orice clasă care conține cel puțin o metodă abstractă este o clasă abstractă.
- Nu.

După cum arată și comentariile, clasa **B** nu poate moșteni clasa **A**, deoarece aceasta a fost declarată ca **sealed**.

Clasa object

Limbajul C# definește o clasă specială, numită **object**, care este clasa de bază implicită pentru toate celelalte clase și pentru toate celelalte tipuri (inclusiv pentru tipurile valorice). Cu alte cuvinte, toate tipurile derivă din clasa **object**. Aceasta înseamnă că o variabilă referință de tipul **object** poate referi un obiect de orice alt tip. Deoarece tablourile sunt și ele implementate prin clase, variabilele de tipul **object** pot de asemenea referi orice tablou. Din punct de vedere tehnic, numele C# **object** este un sinonim pentru **System.Object**, care face parte din biblioteca de clase a arhitecturii .NET.

Clasa **object** definește următoarele metode, care sunt deci disponibile în orice obiect:

Metoda	Semnificația
`public virtual bool Equals(object object)`	Determină dacă obiectul apelant este identic cu cel referit de *object*.
`public static bool Equals(object ob1, object ob2)`	Determină dacă *ob1* și *ob2* referă același obiect.
`protected Finalize()`	Execută acțiunile de distrugere a obiectului, anterioare colectării spațiului ocupat de acesta. În C#, **Finalize()** se accesează prin intermediul destructorilor.
`public virtual int GetHashCode()`	Întoarce un cod de dispersie (hash) asociat obiectului apelant.
`public Type GetType()`	Determină tipul obiectului la momentul execuției.
`protected object MemberwiseClone()`	Efectuează o copie „superficială" a obiectului. Aceasta conține toți membrii obiectului, dar nu și obiectele pe care aceștia le referă.
`public static bool ReferenceEquals(object ob1, object ob2)`	Determină dacă *ob1* și *ob2* referă același obiect.
`public virtual string ToString()`	Întoarce un **string** care descrie obiectul.

Câteva dintre aceste metode impun unele explicații suplimentare. În mod implicit, metoda **Equals(object)** determină dacă obiectul apelant și cel referit de parametrul primit coincid. (Cu alte cuvinte, se determină dacă cele două referințe coincid.) Metoda întoarce **true** dacă obiectele coincid și **false** în caz contrar. Puteți extinde această metodă în clasele create de dumneavoastră. Aceasta vă permite să definiți ce înseamnă relația de egalitate pentru fiecare clasă în parte. De exemplu, puteți defini metoda **Equals(object)** astfel încât să stabilească egalitatea conținutului a două obiecte. Metoda **Equals(object, object)** apelează **Equals(object)** pentru a determina rezultatul.

Metoda **GetHashCode()** întoarce un cod de dispersie (hash) asociat obiectului apelant. Acest cod poate fi utilizat în algoritmii care utilizează dispersia ca metodă de acces la obiectele memorate.

După cum am menționat în modulul 7, pentru a supraîncărca operatorul ==, veți extinde de regulă metodele **Equals(object)** și **GetHashCode()**, deoarece în majoritatea cazurilor, operatorul == și metoda **Equals(object)** trebuie să funcționeze la fel. Dacă extindeți metoda **Equals()**, trebuie de asemenea să extindeți și **GetHashCode()**, pentru asigurarea compatibilității.

Metoda **ToString()** întoarce un **string** care conține o descriere a obiectului pentru care a fost invocată. Această metodă se apelează automat atunci când obiectul este afișat utilizând **WriteLine()**. Multe clase extind această metodă. În acest mod, ele pun la dispoziție o descriere specifică a tipurilor de obiecte pe care le creează. De exemplu:

```
// Prezentarea metodei ToString().
using System;

class MyClass {
  static int count = 0;
  int id;

  public MyClass() {
    id = count;
    count++;
  }

  public override string ToString() {
    return "Instanta MyClass #" + id);
  }
}

class Test {
  public static void Main() {
    MyClass ob1 = new MyClass();
    MyClass ob2 = new MyClass();
    MyClass ob3 = new MyClass();

    Console.WriteLine(ob1);
    Console.WriteLine(ob2);
    Console.WriteLine(ob3);
  }
}
```

Extindem **ToString()**.

Rezultatul programului este prezentat mai jos:

```
Instanta MyClass #0
Instanta MyClass #1
Instanta MyClass #2
```

Împachetare și despachetare

După cum am explicat, toate tipurile din C#, inclusiv tipurile valorice, sunt derivate din **object**. O referință de tipul **object** se poate deci utiliza pentru a referi orice alt tip, inclusiv un tip valoric. Atunci când o referință de tip **object** referă un tip valoric, are loc un fenomen numit *împachetare*. Împachetarea determină memorarea

valorii aparţinând unui tip valoric într-o instanţă a unui obiect. În concluzie, tipul valoric este „împachetat" în interiorul unui obiect. Acest obiect poate fi utilizat ca oricare altul. Împachetarea se produce în toate cazurile în mod automat. Este suficient să atribuiţi o valoare unei referinţe **object**. Restul este realizat de către compilatorul de C#.

Despachetarea este operaţia de determinare a valorii conţinute de un obiect. Aceasta se efectuează utilizând un cast al referinţei **object** la tipul valoric dorit.

Prezentăm mai jos un exemplu simplu, care demonstrează împachetarea şi despachetarea:

```
// Un exemplu simplu cu impachetare/despachetare.
using System;

class BoxDemo {
  public static void Main() {
    int x;
    object obj;

    x = 10;
    obj = x; // impachetam x intr-un obiect

    int y = (int) obj; // despachetam obj intr-o valoare int
    Console.WriteLine(y);
  }
}
```

Se împachetează valoarea lui x.

Despachetăm valoarea.

Acest program va afişa valoarea 10. Remarcaţi că valoarea lui **x** a fost împachetată pur şi simplu după ce a fost atribuită lui **obj**, care este o referinţă de tipul **object**. Valoarea întreagă conţinută de **obj** este regăsită prin castul lui **obj** la **int**.

Iată şi un alt exemplu, ceva mai interesant, de împachetare. În acest exemplu, se transmite o valoare **int** ca parametru metodei **sqr()**, care acceptă un parametru de tipul **object**.

```
// Impachetarea are loc si la transmiterea parametrilor.
using System;

class BoxingDemo {
  public static void Main() {
    int x;

    x = 10;
    Console.WriteLine("Valoarea lui x: " + x);

    // x este automat impachetat atunci cand este transmis lui sqr()
    x = BoxingDemo.sqr(x);
    Console.WriteLine("x la patrat: " + x);
  }

  static int sqr(object o) {
    return (int) o * (int) o;
  }
}
```

Valoarea lui **x** este împachetată la apelul lui **sqr()**.

Rezultatul programului este prezentat mai jos:

```
Valoarea lui x: 10
x la patrat: 100
```

Valoarea lui **x** a fost automat împachetată la transmiterea către metoda **sqr()**.

Împachetarea şi despachetarea permit ca sistemul de tipuri din C# să fie pe deplin unitar. Toate tipurile derivă din **object**. Unei referinţe **object** i se poate deci atribui orice alt tip de referinţă. Împachetarea şi despachetarea rezolvă automat aceste detalii pentru tipurile valorice. Mai mult, deoarece toate tipurile derivă din **object**, orice tip poate accesa metodele clasei **object**. De exemplu, să considerăm următorul program oarecum bizar:

```
// Impachetarea face posibil apelul metodelor pentru o
// valoare !
using System;

class ApelPeValoare {
 public static void Main() {
   Console.WriteLine(10.ToString());
 }
}
```

Perfect corect în C# !

Programul va afişa valoarea 10. Aceasta deoarece metoda **ToString()** întoarce o reprezentare ca **string** a obiectului pentru care este invocată. În acest caz, reprezentarea ca **string** a lui 10 este „10".

Exerciţii la minut

- Dacă o clasă este declarată ca fiind **sealed**, poate fi moştenită?
- Ce este **object**?
- Cum puteţi împiedica o clasă să fie moştenită?

Verificarea cunoştinţelor

1. Poate avea o clasă de bază acces la membrii unei clase derivate? Poate avea o clasă derivată acces la membrii unei clase de bază?
2. Creaţi o clasă derivată din **Forma2D** numită **Cerc**. Includeţi o metodă **aria()** care calculează aria cercului şi un constructor care utilizează **base** pentru iniţializarea porţiunii din clasa de bază **Forma2D**.
3. Cum puteţi împiedica o clasă derivată să acceseze un membru al clasei de bază?
4. Prezentaţi scopurile în care se poate utiliza **base**.

- Nu.
- Clasa **object** este clasa de bază pentru toate tipurile definite în C#.
- Pentru a împiedica moştenirea, clasa trebuie declarată ca **sealed**.

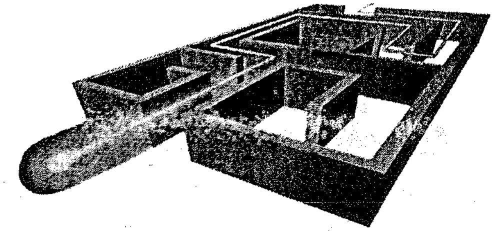
5. Se consideră următoarea ierarhie. În ce ordine se vor apela constructorii acestor clase la instanțierea unui obiect `Gama`?

```
class Alfa { ...
class Beta : Alfa { ...
class Gama : Beta { ...
```

6. O referință de tipul clasei de bază poate referi o instanță a clasei derivate. Precizați ce importanță are acest aspect în ceea ce privește extinderea metodelor.
7. Ce este o clasă abstractă?
8. Cum puteți împiedica o clasă să fie moștenită?
9. Explicați ce rol au moștenirea, extinderea metodelor și clasele abstracte la implementarea conceptului de polimorfism.
10. Care este clasa de bază pentru toate celelalte clase?
11. Explicați fenomenul de împachetare.
12. Cum se pot accesa membrii `protected`?

Modulul 9

Interfețe, structuri și enumerări



Scopurile acestui modul

- **Utilizarea interfețelor**
- **Referințe având ca tip o interfață**
- **Adăugarea proprietăților și indexărilor la interfețe**
- **Moștenirea interfețelor**
- **Utilizarea implementărilor explicite**
- **Studiul structurilor**
- **Studiul enumerărilor**

Acest modul prezintă una dintre cele mai importante facilități din C#: interfețele. O *interfață* definește un set de metode care vor fi implementate de o clasă. Interfețele nu implementează ele însele metode. O interfață reprezintă o construcție pur abstractă, care descrie un set de metode pe care o clasă le va pune la dispoziție, fără a intra însă în detalii de implementare.

Tot în acest modul vom prezenta și alte două tipuri de date din C#: *structurile* și *enumerările*. Structurile sunt similare claselor, fiind însă tratate ca tipuri valorice și nu ca referințe. Enumerările reprezintă liste de constante întregi cu nume. Structurile și enumerările contribuie la varietatea mediului de programare din C#.

Interfețe

În programarea orientată spre obiecte, este uneori foarte util să definiți funcționalitatea unei clase, fără a face însă referire la modul în care se implementează acea funcționalitate. Ați văzut deja un exemplu al acestei abordări: metodele abstracte. Metodele abstracte își definesc numai signatura, nu și implementarea. Clasele derivate trebuie să pună la dispoziție implementări proprii ale tuturor metodelor abstracte definite în clasa de bază. Metodele abstracte își precizează deci *interfața*, dar nu și *implementarea*. Deși metodele și clasele abstracte și-au dovedit utilitatea, este posibil să ducem acest concept cu un pas mai departe. În C#, puteți separa complet interfața unei clase de implementarea ei utilizând cuvântul cheie `interface`.

Din punct de vedere sintactic, interfețele sunt similare cu clasele abstracte. Într-o interfață însă, nici una din metode nu poate avea corp. Aceasta înseamnă că o interfață nu pune la dispoziție nici un fel de implementare. Interfața precizează ce trebuie făcut, dar nu și cum. După ce a fost definită, o interfață poate fi implementată de oricâte clase. Reciproc, o clasă poate implementa oricât de multe interfețe.

Pentru a implementa o interfață, o clasă trebuie să conțină implementările metodelor prezentate în interfață. Fiecare clasă are libertate deplină la stabilirea detaliilor de implementare. Este deci posibil ca două clase să implementeze aceeași interfață în moduri diferite, dar fiecare dintre ele va conține același set de metode. Codul care are cunoștință de existența interfeței poate deci utiliza instanțele oricăreia dintre cele două clase, deoarece interfața cu aceste obiecte este comună. Prin intermediul interfețelor, limbajul C# vă permite să beneficiați la maximum de aspectul „o singură interfață, mai multe metode" al polimorfismului.

Interfețele se declară utilizând cuvântul cheie **interface**. Mai jos, prezentăm o formă simplificată a declarației unei interfețe:

```
interface nume {
    tip-rez nume-metoda1(lista-param);
    tip-rez nume-metoda2(lista-param);
    // ...
    tip-rez nume-metodaN(lista-param);
}
```

Numele interfeței este specificat de către *nume*. Metodele sunt declarate utilizând numai tipul rezultatului lor și signatura. Aceste metode sunt în esență metode abstracte. După cum am precizat, într-o interfață metodele nu pot fi implementate. În consecință, fiecare clasă care include o entitate **interface** trebuie să implementeze toate metodele acesteia. În cadrul interfețelor, metodele sunt în mod implicit **public**, nefiind permisă prezența specificatorilor de acces.

Mai jos, prezentăm un prim exemplu de entitate **interface**. Acesta specifică interfața unei clase care generează o secvență de numere.

```
public interface Sir {
  int getNext(); // intoarce urmatorul numar din sir
  void reset(); // reinitializeaza secventa
  void setStart(int x); // stabileste valoarea initiala
}
```

Această interfață este declarată ca **public**, astfel încât să poată fi implementată de orice clasă din orice program.

Pe lângă signaturile metodelor, interfețele mai pot declara signaturi pentru proprietăți, indexări și evenimente. Evenimentele sunt prezentate în modulul 12; în acest modul, ne vom ocupa numai de metode, proprietăți și indexări. Interfețele nu pot avea variabile membre. Ele nu pot defini constructori, destructori sau metode operator. De asemenea, membrii interfețelor nu pot fi declarați ca **static**.

Implementarea interfețelor

După ce o entitate **interface** a fost definită, una sau mai multe clase o pot implementa. Pentru a implementa o interfață, numele interfeței trebuie să fie specificat după numele clasei, la fel cum este specificată o clasă de bază. Forma generală a unei clase care implementează o interfață este prezentată mai jos:

```
class nume-clasa: nume-interfata {
    // corpul clasei
}
```

Numele interfeței implementate este specificat prin *nume-interfata*.

Atunci când o clasă implementează o interfață, trebuie să o implementeze în totalitate. Clasa nu poate alege să implementeze numai anumite părți ale interfeței.

Clasele pot implementa mai multe interfețe. Pentru aceasta, numele interfețelor implementate trebuie să fie separate prin virgulă. Este posibil ca o clasă să moștenească o clasă de bază și, în același timp, să implementeze una sau mai multe interfețe. În acest caz, numele clasei de bază trebuie să fie primul în lista separată prin virgule.

Metodele care implementează o interfață trebuie declarate ca **public**. Aceasta deoarece metodele sunt în mod implicit publice în cadrul interfeței, deci și implementările lor trebuie să fie de asemenea publice. De asemenea, signatura tipurilor din implementarea metodei trebuie să se potrivească exact cu signatura de tipuri precizată în definiția **interface**.

Mai jos, prezentăm un exemplu de implementare a interfeței **Sir** pe care am prezentat-o mai devreme. Exemplul creează o clasă numită **DoiInDoi**, care generează o secvență de numere în care fiecare este cu 2 mai mare decât precedentul.

```
// Implementarea interfetei Sir.
class DoiInDoi : Sir {    <-- Implementează interfața Sir.
  int start;
  int val;

  public DoiInDoi() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
```

```
      val = x;
    }
  }
```

După cum puteți observa, clasa **DoiInDoi** implementează toate cele trei metode definite de interfața **Sir**. Cum am mai explicat, aceasta este necesar, deoarece o clasă nu poate crea doar o implementare parțială a unei interfețe.

În continuare, iată o clasă care utilizează clasa **DoiInDoi**:

```
// Utilizeaza clasa DoiInDoi.
using System;

class SirDemo {
  public static void Main() {
    DoiInDoi ob = new DoiInDoi();

    for(int i = 0; i < 5; i++)
      Console.WriteLine("Valoarea urmatoare este " +
                         ob.getNext());

    Console.WriteLine("\nReinitializare");
    ob.reset();
    for(int i = 0; i < 5; i++)
      Console.WriteLine("Valoarea urmatoare este " +
                         ob.getNext());

    Console.WriteLine("\nInitializare la 100");
    ob.setStart(100);
    for(int i = 0; i < 5; i++)
      Console.WriteLine("Valoarea urmatoare este " +
                         ob.getNext());
  }
}
```

Pentru a compila programul **SirDemo**, trebuie să includeți clasele **Sir**, **DoiInDoi** și **SirDemo** la compilare. Compilatorul va compila automat toate cele trei fișiere pentru a crea executabilul final. De exemplu, dacă ați denumit aceste fișiere **Sir.cs**, **DoiInDoi.cs** și **SirDemo.cs**, puteți folosi următoarea linie de comandă pentru a compila programul:

```
>csc Sir.cs DoiInDoi.cs SirDemo.cs
```

Dacă utilizați mediul integrat Visual C++, adăugați pur și simplu toate cele trei fișiere la proiectul dumneavoastră C#. O altă soluție, perfect corectă, este să includeți toate cele trei clase în același fișier.

Rezultatul afișat de acest program este prezentat mai jos:

```
Valoarea urmatoare este 2
Valoarea urmatoare este 4
Valoarea urmatoare este 6
Valoarea urmatoare este 8
Valoarea urmatoare este 10

Reinitializare
Valoarea urmatoare este 2
Valoarea urmatoare este 4
Valoarea urmatoare este 6
Valoarea urmatoare este 8
Valoarea urmatoare este 10

Initializare la 100
Valoarea urmatoare este 102
Valoarea urmatoare este 104
Valoarea urmatoare este 106
Valoarea urmatoare este 108
Valoarea urmatoare este 110
```


Clasele care implementează interfețe pot defini – și de regulă chiar definesc – membri proprii, pe lângă cei ai interfețelor. De exemplu, versiunea următoare a clasei **DoiInDoi** adaugă metoda **getPrevious()**, care întoarce valoarea anterioară:

```
// Implementarea interfetei Sir, adaugand metoda
// proprie getPrevious().
class DoiInDoi : Sir {
  int start;
  int val;
  int prev;

  public DoiInDoi() {
    start = 0;
    val = 0;
    prev = -2;
  }

  public int getNext() {
    prev = val;
    val += 2;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
    prev = -2;
  }

  public void setStart(int x) {
    start = x;
    val = x;
    prev = x - 2;
  }

  // O metoda care nu este specificata de Sir.
  int getPrevious() {   <-- Adăugăm o metodă care nu este definită de interfața Sir.
    return prev;
  }
}
```

Remarcați că adăugarea metodei **getPrevious()** necesită o modificare a implementărilor metodelor definite de interfața **Sir**. Deoarece interfața cu aceste metode rămâne aceeași, modificarea păstrează continuitatea și nu influențează codul deja existent. Acesta este unul dintre avantajele interfețelor.

După cum am explicat, o interfață poate fi implementată de oricâte clase. De exemplu, mai jos prezentăm o clasă numită **TreiInTrei**, care generează o progresie aritmetică de rație 3:

```
// O alta implementare a interfetei Sir.
class TreiInTrei : Sir {
  int start;
  int val;

  public TreiInTrei() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 3;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

O altă implementare a interfeței **Sir**.

Exerciții la minut

- În general, care este rolul unei interfețe?
- Care sunt elementele care pot fi membre ale unei interfețe?
- Cum poate o clasă implementa o interfață?

Referințe având ca tip o interfață

Deși pare surprinzător la prima vedere, veți învăța că puteți declara o variabilă referință având ca tip o interfață. Cu alte cuvinte, puteți crea o variabilă de tipul referință către o interfață. O astfel de variabilă poate referi orice obiect care implementează interfața. La apelul unei metode prin intermediul referinței către interfață, se va executa versiunea metodei care este implementată de obiectul

- O interfață definește funcționalitatea unei clase, dar nu și modul în care aceasta este implementată.
- Metodele, proprietățile, indexările și evenimentele pot fi membri ai unei interfețe.
- Pentru ca o clasă să implementeze o interfață, numele interfeței trebuie specificat după numele clasei, iar clasa trebuie să conțină o implementare pentru toți membrii interfeței.

apelant. Acest mecanism este similar cu utilizarea unei referințe de tipul clasei de bază pentru accesul la o instanță a unei clase derivate, mecanism prezentat în modulul 8.

Exemplul următor prezintă utilizarea unei referințe către o interfață. Programul utilizează aceeași variabilă referință având ca tip o interfață pentru a apela metodele interfeței asupra instanțelor claselor **DoiInDoi** si **TreiInTrei**.

```
// Prezinta referintele catre o interfata.
using System;

public interface Sir {
  int getNext(); // intoarce urmatorul numar din sir
  void reset(); // reinitializeaza secventa
  void setStart(int x); // stabileste valoarea initiala
}

// O prima implementare a interfetei Sir.
class DoiInDoi : Sir {
  int start;
  int val;

  public DoiInDoi() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 2;
    return val;
  }

  public void reset() {
    start = 0;
    val = 0;
  }

  public void setStart(int x) {
    start = x;
    val = x;
  }
}

// O alta implementare a interfetei Sir.
class TreiInTrei : Sir {
  int start;
  int val;

  public TreiInTrei() {
    start = 0;
    val = 0;
  }

  public int getNext() {
    val += 3;
    return val;
  }
```

```
    public void reset() {
      start = 0;
      val = 0;
    }

    public void setStart(int x) {
      start = x;
      val = x;
    }
}

class SirDemo2 {
    public static void Main() {
      DoiInDoi doiOb = new DoiInDoi();
      TreiInTrei treiOb = new TreiInTrei();
      Sir ob;

      for(int i = 0; i < 5; i++) {
        ob = doiOb;
        Console.WriteLine("Urmatoarea valoare din sirul de
                          ratie 2 este " + ob.getNext());
        ob = treiOb;
        Console.WriteLine("Urmatoarea valoare din sirul de
                          ratie 3 este " + ob.getNext());
      }
    }
}
```

Accesul la un obiect prin intermediul unei referințe către o interfață.

În **Main()**, variabila **ob** este declarată ca referință la interfața **Sir**. Aceasta înseamnă că poate fi utilizată pentru a memora referințe către orice tip de obiecte care implementează interfața **Sir**. În acest caz, este utilizată pentru a referi **doiOb** și **treiOb**, obiecte de tipul **DoiInDoi** și respectiv **TreiInTrei**, tipuri care implementează ambele interfața **Sir**. O variabilă de tipul referință la o interfață cunoaște numai metodele din declarația **interface**. Variabila **ob** nu poate fi deci utilizată pentru accesul la variabilele și metodele care nu sunt cuprinse în interfață.

Proiectul 9-1: Crearea unei interfețe de coadă

Pentru a vedea puterea interfețelor la lucru, să considerăm un exemplu practic. În primele module, ați dezvoltat o clasă numită **Queue**, care implementează o coadă simplă de caractere, de lungime fixată. Există însă multe alte modalități de a implementa o coadă. De exemplu, coada poate avea lungimea fixată sau poate „crește". Coada poate fi *liniară*, dacă este utilizată secvențial, până la epuizare, sau *circulară*, caz în care se pot adăuga elemente atât timp cât se extrag alte elemente. Coada poate fi de asemenea memorată într-un tablou, într-o listă înlănțuită, într-un arbore binar etc. Indiferent de implementarea cozii, interfața sa rămâne aceeași, fiind definită de metodele **put()** și **get()** în mod independent de detaliile de implementare. Deoarece interfața unei cozi este separată de implementarea sa, cel mai simplu este să definiți o interfață de coadă, lăsând definirea detaliilor specifice în sarcina fiecărei implementări în parte.

În cadrul acestui proiect, veți crea interfața pentru o coadă de caractere și trei implementări ale acesteia. Toate cele trei implementări vor utiliza un tablou pentru memorarea caracterelor. Prima dintre cozi va fi coada liniară de lungime fixată pe care ați dezvoltat-o anterior. Cea de-a doua implementare va fi o coadă circulară. Într-o coadă circulară, atunci când se ajunge la sfârșitul tabloului utilizat pentru memorarea datelor, indicii **get** și **put** revin automat la început. În consecință, coada circulară va putea memora un număr oricât de mare de elemente, cu condiția ca elementele să fie și extrase din coadă. Ultima implementare va crea o coadă dinamică. Aceasta poate crește cu cât are nevoie atunci când dimensiunea sa este depășită.

Pas cu pas

1. Creați un fișier numit **ICharQ.cs**, în care includeți următoarea definiție:

```
// Interfata unei cozi de caractere.
public interface ICharQ {
  // Insereaza un caracter in coada.
  void put(char ch);
  // Extrage un caracter din coada.
  char get();
}
```

După cum observați, această interfață este extrem de simplă, constând numai din două metode. Toate clasele care vor implementa interfața **ICharQ** trebuie să implementeze aceste metode.

2. Creați un alt fișier, numit **IQDemo.cs**.
3. Adăugați pentru început în fișierul **IQDemo.cs** clasa **FixedQueue**, prezentată mai jos:

```
/*
    Proiectul 9-1
    Utilizarea interfetei ICharQ.
*/
using System;

// O clasa care reprezinta o coada de caractere de lungime fixata.
 class FixedQueue : ICharQ {
   char [] q; // tabloul care memoreaza coada
   int putloc, getloc; // indicii put si get

   // Construieste o coada vida de dimensiune data.
   public FixedQueue(int size) {
     q = new char[size + 1]; // alocam memorie pentru coada
     putloc = getloc = 0;
   }

   // Inseram un caracter in coada.
   public void put(char ch) {
     if(putloc == q.Length - 1) {
       Console.WriteLine("- Coada este plina.");
       return;
     }

     putloc++;
     q[putloc] = ch;
   }

   // Extrage un caracter din coada.
   public char get() {
     if(getloc == putloc) {
```

```
      Console.WriteLine("- Coada este vida.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
```

Această implementare a interfeței `ICharQ` este adaptată după clasa `Queue` prezentată în modulul 5 și ar trebui să vă fie cunoscută.

4. Adăugați tot în fișierul `IQDemo.cs` clasa `CircularQueue` prezentată mai jos. Aceasta implementează o coadă circulară de caractere.

```
// O coada circulara.
class CircularQueue : ICharQ {
  char [] q; // tabloul care memoreaza coada
  int putloc, getloc; // indicii put si get

  // Construieste o coada vida de dimensiune data.
  public CircularQueue(int size) {
    q = new char[size + 1];   // alocam memorie pentru coada
    putloc = getloc = 0;
  }

  // Inseram un caracter in coada.
  public void put(char ch) {
    /* Coada este plina daca putloc este cu o unitate mai mic
       decat getloc, sau daca putloc este la sfarsitul
       tabloului, iar getloc la inceput. */

    if(putloc + 1 == getloc |
       ((putloc == q.Length - 1) & (getloc == 0))) {
      Console.WriteLine("- Coada este plina.");
      return;
    }
    putloc++;
    if(putloc == q.Length) putloc = 0;  // intoarcere inapoi
    q[putloc] = ch;
  }

  // Extrage un caracter din coada.
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine("- Coada este vida.");
      return (char) 0;
    }
    getloc++;
    if(getloc == q.Length) getloc = 0; // intoarcere inapoi
    return q[getloc];
  }
}
```

Coada circulară funcționează reutilizând spațiul din tablou, care este eliberat la extragerea elementelor. Acest tip de coadă poate deci memora un număr nelimitat de elemente, cu condiția ca elementele să fie și extrase din coadă. Deși sunt simple din punct de vedere conceptual – indicii corespunzători revin la valoarea 0 atunci când se atinge sfârșitul tabloului – condițiile de margine vă pot induce în eroare la început. Într-o coadă circulară, condiția de coadă plină se activează nu atunci când se atinge sfârșitul tabloului, ci atunci când memorarea unui nou element ar conduce la suprascrierea unui element care nu a fost încă extras. Metoda `put()` trebuie deci să verifice mai multe condiții pentru a determina dacă s-a produs umplerea cozii. După cum arată și comentariile din program, coada este plină fie când `putloc` este cu o unitate mai mic decât `getloc`, fie dacă `putloc` indică sfârșitul tabloului, iar `getloc` începutul acestuia. Ca și mai înainte, coada este vidă atunci când `getloc` și `putloc` au valori egale.

5. În fine, inserați în fișierul `IQDemo.cs` clasa `DynQueue`, prezentată în continuare. Aceasta implementează o coadă care poate „crește", mărindu-și dimensiunea atunci când spațiul său se epuizează.

```
// O coada dinamica.
class DynQueue : ICharQ {
  char [] q; // tabloul care memoreaza coada
  int putloc, getloc; // indicii put si get

  // Construieste o coada vida de dimensiune data.
  public DynQueue(int size) {
    q = new char[size + 1]; // alocam memorie pentru coada
    putloc = getloc = 0;
  }

  // Inseram un caracter in coada.
  public void put(char ch) {
    if(putloc == q.Length - 1) {
      // Marim dimensiunea cozii.
      char [] t = new char[q.Length * 2];
      // copiem elementele in noua coada.
      for(int i = 0; i < q.Length; i++)
        t[i] = q[i];

      q = t;
    }

    putloc++;
    q[putloc] = ch;
  }

  // Extrage un caracter din coada.
  public char get() {
    if(getloc == putloc) {
      Console.WriteLine("- Coada este vida.");
      return (char) 0;
    }

    getloc++;
    return q[getloc];
  }
}
```

În această implementare, atunci când coada este plină, încercarea de a memora încă un element determină realocarea tabloului de elemente cu o dimensiune dublă față de cea inițială. Conținutul curent al cozii este apoi copiat în acest tablou, iar variabila `q` primește ca valoare o referință către noul tablou.

6. Pentru a demonstra toate cele trei implementări ale interfeței `ICharQ`, introduceți și clasa următoare în fișierul `IQDemo.cs`. Aceasta utilizează o referință de tipul `ICharQ` pentru a accesa toate cele trei tipuri de cozi.

```
// Demonstreaza utilizarea cozilor.
class IQDemo {
  public static void Main() {
    FixedQueue q1 = new FixedQueue(10);
    DynQueue q2 = new DynQueue(5);
    CircularQueue q3 = new CircularQueue(10);

    ICharQ iQ;

    char ch;
    int i;

    iQ = q1;
    // Inseram cateva caractere in coada de lungime fixata.
    for(i = 0; i < 10; i++)
      iQ.put((char)('A' + i));

    // afisam coada de lungime fixata.
    Console.WriteLine("Coada de lungime fixata: ");
    for(i = 0; i < 10; i++) {
      ch = iQ.get();
      Console.Write(ch);
    }
    Console.WriteLine();

    iQ = q2;
    // Inseram cateva caractere in coada dinamica.
    for(i = 0; i < 10; i++)
      iQ.put((char)('Z' - i));

    // afisam coada dinamica.
    Console.WriteLine("Coada dinamica: ");
    for(i = 0; i < 10; i++) {
      ch = iQ.get();
      Console.Write(ch);
    }

    Console.WriteLine();

    iQ = q3;
    // Inseram cateva caractere in coada circulara.
    for(i = 0; i < 10; i++)
      iQ.put((char)('A' + i));

    // afisam coada circulara.
    Console.WriteLine("Coada circulara: ");
    for(i = 0; i < 10; i++) {
      ch = iQ.get();
      Console.Write(ch);
    }

    Console.WriteLine();

    // Inseram mai multe caractere in coada circulara.
    for(i = 10; i < 20; i++)
      iQ.put((char)('A' + i));

    // afisam din nou coada circulara.
    Console.WriteLine("Coada circulara: ");
    for(i = 0; i < 10; i++) {
      ch = iQ.get();
      Console.Write(ch);
    }

    Console.WriteLine("\n Memoram si extragem date din coada circulara.");

    // Extragem caractere din coada circulara.
    for(i = 0; i < 20; i++) {
      iQ.put((char)('A' + i));
      ch = iQ.get();
      Console.Write(ch);
    }

  }
}
```


7. Compilați programul, incluzând ambele fișiere `ICharQ.cs` și `IQDemo.cs`.
8. Rezultatul acestui program este prezentat mai jos:

```
Coada de lungime fixata: ABCDEFGHIJ
Coada dinamica: ZYXWVUTSRQ
Coada circulara: ABCDEFGHIJ
Coada circulara: KLMNOPQRST
Memoram si extragem date din coada circulara.
ABCDEFGHIJKLMNOPQRST
```

9. Iată câteva îmbunătățiri pe care le puteți încerca pe cont propriu. Creați o versiune circulară a clasei `DynQueue`. Adăugați o metodă `reset()` care inițializează o coadă. Creați o metodă `static` care copiază conținutul unei cozi în altă coadă.

Utilizarea proprietăților în interfețe

Ca și metodele, proprietățile se pot specifica în cadrul unei interfețe fără a include corpul. Forma generală a specificației unei proprietăți este prezentată mai jos:

```
// proprietatea
tip nume-prop {
   get;
   set;
}
```

Desigur, în cazul proprietăților accesibile numai la citire, respectiv numai la scriere, va apărea numai accesorul `get`, respectiv numai accesorul `set`.

Iată o variantă modificată a interfeței `Sir` și a clasei `DoiInDoi` care utilizează o proprietate pentru a obține și pentru a modifica următorul element din șir:

```
// Utilizarea unei proprietati intr-o interfata.
using System;
```

```
public interface Sir {
  // o proprietate
  int next {
    get; // intoarce urmatorul numar din sir
    set; // modifica urmatorul numar
  }
}

// Implementarea interfetei Sir.
class DoiInDoi : Sir {
  int val;

  public DoiInDoi() {
    val = 0;
  }

  // intoarce sau modifica valoarea
  public int next {
    get {
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }
}

// Utilizeaza proprietatea din interfata.
class SirDemo3 {
  public static void Main() {
    DoiInDoi ob = new DoiInDoi();

    // acces la sir prin intermediul unei proprietati
    for(int i = 0; i < 5; i++)
      Console.WriteLine("Valoarea urmatoare este " + ob.next);

    Console.WriteLine("\nInitializare la 21");
    ob.next = 21;
    for(int i = 0; i < 5; i++)
    Console.WriteLine("Valoarea urmatoare este " + ob.next);
  }
}
```

Declararea unei proprietăți în interfața **Sir**.

Implementarea proprietății.

Rezultatul programului este prezentat în cele ce urmează:

```
Valoarea urmatoare este 2
Valoarea urmatoare este 4
Valoarea urmatoare este 6
Valoarea urmatoare este 8
Valoarea urmatoare este 10

Initializare la 21
Valoarea urmatoare este 23
Valoarea urmatoare este 25
Valoarea urmatoare este 27
Valoarea urmatoare este 29
Valoarea urmatoare este 31
```


Utilizarea indexărilor în interfețe

O indexare declarată în cadrul unei interfețe are următoarea formă generală:

```
// indexare in interfata
tip-element this[int index] {
   get;
   set;
}
```

Ca și mai înainte, indexările accesibile numai la citire, respectiv numai la scriere, vor defini numai accesorul **get**, respectiv accesorul **set**.

În continuare prezentăm o altă versiune a interfeței **Sir**, care adaugă o indexare accesibilă numai la citire, care întoarce elementul cu rangul **i** din șir.

```
// Adaugarea unei indexari la o interfata.
using System;

public interface Sir {
  // o proprietate
  int next {
    get; // intoarce urmatorul numar din sir
    set; // modifica urmatorul numar
  }

  // o indexare
  int this[int index] {
    get; // intoarce elementul cu rangul specificat din sir
  }
}

// Implementarea interfetei Sir.
class DoiInDoi : Sir {
  int val;

  public DoiInDoi() {
    val = 0;
  }

  // intoarce sau modifica valoarea
  public int next {
    get {
      val += 2;
      return val;
    }
    set {
      val = value;
    }
  }

  // intoarce o valoare utilizand un index
  public int this[int index] {
    get {
      val = 0;
      for(int i = 0; i < index; i++)
        val += 2;
```

Declararea unei indexări accesibile numai la citire.

Implementarea indexării.

```
      return val;
    }
  }
}

// Utilizeaza indexarea din interfata.
class SirDemo4 {
  public static void Main() {
    DoiInDoi ob = new DoiInDoi();

    // acces la sir prin intermediul unei proprietati
    for(int i = 0; i < 5; i++)
      Console.WriteLine("Valoarea urmatoare este " + ob.next);

    Console.WriteLine("\nInitializare la 21");
    ob.next = 21;
    for(int i = 0; i < 5; i++)
      Console.WriteLine("Valoarea urmatoare este " + ob.next);

    Console.WriteLine("Initializare la 0");
    ob.next = 0;

    // accesul la sir utilizand indexarea
    for(int i = 0; i < 5; i++)
      Console.WriteLine("Valoarea urmatoare este " + ob[i]);
  }
}
```

Rezultatul programului este prezentat mai jos:

```
Valoarea urmatoare este 2
Valoarea urmatoare este 4
Valoarea urmatoare este 6
Valoarea urmatoare este 8
Valoarea urmatoare este 10

Initializare la 21
Valoarea urmatoare este 23
Valoarea urmatoare este 25
Valoarea urmatoare este 27
Valoarea urmatoare este 29
Valoarea urmatoare este 31

Initializare la 0
Valoarea urmatoare este 0
Valoarea urmatoare este 2
Valoarea urmatoare este 4
Valoarea urmatoare este 6
Valoarea urmatoare este 8
```


Exerciții la minut

- Poate o variabilă referință, având tipul dat de o interfață, referi un obiect care implementează acea interfață?
- Pot fi proprietățile din interfețe accesibile numai la citire?
- Au specificațiile indexărilor și proprietăților din interiorul unei interfețe un corp propriu?

Interfețele pot fi moștenite

O interfață poate moșteni o altă interfață. Sintaxa este comună cu cea care se utilizează la moștenirea claselor. Dacă o clasă implementează o interfață care moștenește o altă interfață, clasa trebuie să conțină implementări pentru toți membrii definiți pe lanțul de moștenire al interfeței. În continuare, prezentăm un exemplu:

```
// O interfata poate mosteni o alta interfata.
using System;

public interface A {
  void met1();
  void met2();
}

// interfata B include met1() si met2(), la care adauga met3().
public interface B : A {   <-- Interfața B moștenește pe A.
  void met3();
}

// aceasta clasa trebuie sa implementeze toate metodele din A si B
class MyClass : B {
  public void met1() {
    Console.WriteLine("Implementarea lui met1().");
  }

  public void met2() {
    Console.WriteLine("Implementarea lui met2().");
  }

  public void met3() {
    Console.WriteLine("Implementarea lui met3().");
  }
}

class IFExtend {
  public static void Main() {
    MyClass ob = new MyClass();

    ob.met1();
```

- Da.
- Da.
- Nu.

```
        ob.met2();
        ob.met3();
    }
}
```

Sfatul expertului

Întrebare: Atunci când o interfață moștenește o alta, este posibil să declarăm un membru în interfața derivată care să ascundă un membru al interfeței de bază?

Răspuns: Da. Dacă un membru al interfeței derivate are aceeași signatură cu membrul omonim din interfața de bază, numele din cadrul interfeței de bază este ascuns. Ca și în cazul moștenirii claselor, această situație va genera un mesaj de avertisment dacă membrul interfeței derivate nu este precedat de cuvântul cheie **new**.

Cu rol de experiment, puteți încerca să eliminați implementarea metodei **met1()** din clasa **MyClass**. Aceasta va produce o eroare la compilare. Cum am precizat mai devreme, orice clasă care implementează o interfață trebuie să implementeze toate metodele definite de acea interfață, inclusiv pe cele moștenite de la alte interfețe.

Implementări explicite

Atunci când implementați un membru al unei interfețe, este posibil să specificați *numele complet calificat*, prefixând numele membrului cu numele interfeței. În acest mod, se creează o *implementare explicită a membrului interfeței*, sau pe scurt *implementare explicită*. De exemplu, dându-se interfața:

```
interface IMyIF {
    int myMeth(int x);
}
```

este corect să implementăm interfața **IMyIF** după cum urmează:

```
class MyClass : IMyIF {
    int IMyIF.myMeth(int x) {
        return x / 3;
    }
}
```

După cum se observă, la implementarea membrului **myMeth()** al interfeței **IMyIF**, am specificat numele complet calificat, care include și numele interfeței.

Există două motive pentru care implementările explicite ale membrilor unei interfețe sunt necesare. Mai întâi, este posibil ca o clasă să implementeze două interfețe distincte, care declară fiecare metode având același nume și aceeași signatură de tipuri. Specificarea numelor complet calificate elimină ambiguitatea în această situație. În al doilea rând, implementarea unei metode cu numele complet calificat este echiva-

lentă cu o implementare privată, care nu este vizibilă pentru codul din afara clasei. Să studiem fiecare dintre aceste motive cu ajutorul unui exemplu.

Programul din continuare conține o interfață numită **IPar**, care definește două metode: **estePar()** și **esteImpar()**, care stabilesc dacă un număr este par, respectiv impar. Clasa **MyClass** implementează apoi interfața **IPar**. Implementarea metodei **esteImpar()** este explicită.

```
// Implementarea explicita a unui membru al unei interfete.
using System;

interface IPar {
    bool estePar(int x);
    bool esteImpar(int x);
}

class MyClass : IPar {
    // implementare explicita
    bool IPar.esteImpar(int x) {
        if((x % 2) != 0) return true;
        else return false;
    }

    // implementare normala
    public bool estePar(int x) {
        IPar o = this; // referinta la obiectul apelant

        return !o.esteImpar(x);
    }
}

class Demo {
    public static void Main() {
        MyClass ob = new MyClass();
        bool result;

        result = ob.estePar(4);
        if(result) Console.WriteLine("4 este par.");
        else Console.WriteLine("3 este impar.");

        // result = ob.esteImpar(5); // Eroare, nu este vizibila
    }
}
```

Implementare explicită a metodei **esteImpar()**; aceasta devine practic privată.

Deoarece metoda **esteImpar()** este implementată explicit, ea nu este vizibilă în afara clasei **MyClass**. Metoda a devenit practic privată. În interiorul clasei **MyClass**, **esteImpar()** poate fi accesată numai utilizând o referință de tipul interfeței. Din acest motiv, în implementarea metodei **estePar()**, ea este invocată prin intermediul lui **o**.

Iată și un exemplu de clasă care implementează două interfețe care declară ambele câte o metodă numită **met()**. În această situație, implementarea explicită se utilizează pentru eliminarea ambiguității.

```
// Utilizarea implementarii explicite pentru eliminarea ambiguitatii.
using System;
```

```
interface IMyIF_A {
  int met(int x);
}

interface IMyIF_B {
  int met(int x);
}

// Clasa MyClass implementeaza ambele interfete.
class MyClass : IMyIF_A, IMyIF_B {
  IMyIF_A a_ob;
  IMyIF_B b_ob;

  // implementam explicit cele doua metode met()
  int IMyIF_A.met(int x) {
    return x + x;
  }

  int IMyIF_B.met(int x) {
    return x * x;
  }

  // apelam met() printr-o referinta de tipul interfetei.
  public int metA(int x) {
    a_ob = this;
    return a_ob.met(x); // apel la IMyIF_A.met()
  }

  public int metB(int x) {
    b_ob = this;
    return b_ob.met(x); // apel la IMyIF_B.met()
  }
}

class FQIFNames {
  public static void Main() {
    MyClass ob = new MyClass();

    Console.Write("Apel la IMyIF_A.met(): ");
    Console.WriteLine(ob.metA(3));

    Console.Write("Apel la IMyIF_B.met(): ");
    Console.WriteLine(ob.metB(3));
  }
}
```

Signaturile pentru aceste două metode sunt identice.

Implementarea explicită îndepărtează ambiguitatea.

Rezultatul acestui program este prezentat mai jos:

```
Apel la IMyIF_A.met(): 6
Apel la IMyIF_B.met(): 9
```

Examinând programul, remarcaţi mai întâi că metoda **met()** are aceeaşi signatură în amândouă interfeţele **IMyIF_A** şi **IMyIF_B**. Atunci când **MyClass** implementează ambele interfeţe, trebuie deci să implementeze explicit fiecare dintre cele două metode, utilizând numele complet calificat la implementare. Deoarece singura modalitate de apel pentru metodele implementate explicit este prin intermediul referinţelor către interfeţe, **MyClass** creează două astfel de referinţe, una pentru **IMyIF_A** şi cealaltă pentru **IMyIF_B**. Clasa defineşte apoi două metode proprii care apelează metodele definite în interfeţe, eliminând astfel ambiguitatea.

Exerciţii la minut

- Dacă interfaţa A este moştenită de interfaţa B, iar aceasta este implementată de clasa C, trebuie C să implementeze toţi membrii din A şi B sau numai pe cei definiţi în B?
- Precizaţi două motive pentru care este necesar să utilizaţi o implementare explicită a unei interfeţe.

Structuri

După cum ştiţi deja, clasele sunt tipuri referinţă. Aceasta înseamnă că obiectele claselor sunt accesate prin intermediul referinţelor, spre deosebire de tipurile valorice, care sunt utilizate direct. Există însă unele cazuri în care este util să puteţi accesa un obiect în mod direct. Unul dintre motive îl constituie eficienţa. Accesul la obiecte prin intermediul referinţelor adaugă o întârziere la fiecare operaţie de acest gen şi, mai mult, consumă şi spaţiu de memorie suplimentar. Pentru obiectele foarte mici, acest spaţiu consumat suplimentar devine semnificativ. Pentru a rezolva aceste probleme, limbajul C# pune la dispoziţie structurile. O *structură* este similară unei clase, fiind însă un tip valoric şi nu un tip referinţă.

Structurile pot fi declarate utilizând cuvântul cheie **struct** şi sunt similare, din punct de vedere sintactic, cu clasele. Forma generală a unei declaraţii **struct** este prezentată în continuare:

```
struct nume: interfete {
  // declaratii de membri
}
```

Numele structurii este specificat prin *nume*.

Structurile nu pot moşteni alte structuri sau alte clase şi nici nu pot fi utilizate ca bază pentru alte structuri sau clase. Ele pot însă implementa una sau mai multe interfeţe. Acestea sunt specificate după numele structurii sub forma unei liste separate prin virgule.

Ca şi în cazul claselor, membrii structurilor pot fi metode, câmpuri, indexări, proprietăţi, metode operator şi evenimente. Structurile pot de asemenea defini constructori, dar nu şi destructori. Nu puteţi însă defini constructori impliciţi (fără

- Clasa C trebuie să implementeze toţi membrii specificaţi în cadrul ierarhiei de interfeţe.
- Implementările explicite se utilizează pentru a elimina ambiguitatea şi pentru a evita expunerea unui membru al unei interfeţe.

parametri) pentru structuri. Aceasta se întâmplă deoarece un constructor implicit este definit în mod automat pentru toate structurile şi acesta nu poate fi modificat.

Un obiect structură poate fi creat utilizând **new** în acelaşi fel ca şi pentru o instanţă a unei clase, dar nu este obligatoriu. Dacă nu se utilizează **new**, obiectul este creat, fără a fi însă iniţializat. În acest caz, trebuie să efectuaţi iniţializarea structurii manual.

Iată un exemplu care utilizează o structură:

```
// Utilizarea unei structuri.
using System;

// Definim o structura.
struct cont {   <-- Defineşte structura.
  public string nume;
  public double sold;

  public cont(string n, double s) {
    nume = n;
    sold = s;
  }
}

// Utilizam structura cont.
class StructDemo {
  public static void Main() {
    cont c1 = new cont("Ion", 1232.22); // constructor explicit
    cont c2 = new cont();  // constructor implicit
    cont c3;   // fara constructor

    Console.WriteLine(c1.nume + " are un sold de " + c1.sold);
    Console.WriteLine();

    if(c2.nume == null) Console.WriteLine("c2.nume este null.");
    Console.WriteLine("c2.sold este " + c2.sold);
    Console.WriteLine();

    // c3 trebuie initializat inainte de utilizare
    c3.nume = "Maria";
    c3.sold = 999.33;
    Console.WriteLine(c3.nume + " are un sold de " + c3.sold);
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Ion are un sold de 1232.22

c2.nume este null.
c2.sold este 0

Maria are un sold de 999.33
```


Sfatul expertului

Întrebare: Se ştie că şi limbajul C++ dispune de structuri şi utilizează cuvântul cheie struct. Sunt structurile din C# similare cu cele din C++?

Răspuns: Nu. În C++, **struct** defineşte un tip clasă. În concluzie, în C++ **struct** şi **class** sunt aproape echivalente. (Diferenţa constă în accesul implicit la membri, care este privat pentru **class** şi public pentru **struct**.) În C#, **struct** defineşte un tip valoric, iar **class** defineşte un tip referinţă.

După cum arată acest program, structurile se pot iniţializa fie prin utilizarea lui **new**, care invocă mai departe un constructor, fie prin simpla declarare a unui obiect. Dacă se utilizează **new**, câmpurile structurii vor fi iniţializate fie de către constructorul implicit cu valorile implicite corespunzătoare, fie de către un constructor definit de utilizator. Dacă nu se utilizează **new**, atunci obiectul nu este iniţializat şi nu poate fi utilizat până când câmpurile sale nu primesc valori.

Enumerări

O *enumerare* este o mulţime de constante întregi cu nume, care specifică toate valorile posibile pe care le pot lua variabilele care au ca tip enumerarea. Enumerările sunt frecvent întâlnite în viaţa de zi cu zi. De exemplu, o enumerare a monedelor utilizate în România este:

1 leu, 5 lei, 10 lei, 20 lei, 50 lei, 100 lei, 500 lei, 1000 lei

Un tip enumerare se declară utilizând cuvântul cheie **enum**. Forma generală a unei enumerări este:

enum *nume* { *lista-constante* };

Mai sus, numele tipului enumerare este specificat de către *nume*. *Lista-constante* este o listă de identificatori, aceştia fiind separaţi prin virgule.

Secvenţa următoare de cod defineşte o enumerare numită **monede**:

```
enum monede { unLeu, cinciLei, zeceLei, douazeciLei,
cincizeciLei, oSutaLei, cinciSuteLei, oMieLei };
```

Aspectul esenţial care trebuie clarificat privitor la enumerări este că fiecare simbol dintr-o enumerare reprezintă o valoare întreagă. Astfel, aceste constante simbolice se pot utiliza oriunde în locul unei valori întregi. Fiecare constantă simbolică primeşte o valoare cu o unitate mai mare faţă de constanta precedentă din enumerare. În mod implicit, valoarea primei constante simbolice din enumerare este 0.

Membrii unei enumerări pot fi accesaţi prin intermediul numelui tipului enumerare, cu ajutorul operatorului punct. De exemplu, instrucţiunea:

```
Console.WriteLine("unLeu: " + monede.unLeu + " cinciLei: " + monede.cinciLei);
```

afişează:

```
unLeu: 0 cinciLei: 1
```

Mai jos, prezentăm un program care utilizează enumerarea **monede**:

```
// Utilizarea unei enumerari.
using System;

class EnumDemo {
  enum monede { unLeu, cinciLei, zeceLei, douazeciLei,
                cincizeciLei, oSutaLei, cinciSuteLei, oMieLei };

  public static void Main() {
    string [] nume = { "1 leu",
      "5 lei",
      "10 lei",
      "20 lei",
      "50 lei",
      "100 lei",
      "500 lei",
      "1000 lei"
    };

    monede i;   // declaram o variabila enum

    // utilizam i pentru a itera prin enumerare
    for(i = monede.unLeu; i <= monede.oMieLei; i++)
      Console.WriteLine(nume[(int)i] + " are valoarea: " + i);
  }
}
```

O variabilă de tipul **monede** poate controla o buclă **for**.

Rezultatul acestui program este prezentat mai jos:

```
1 leu are valoarea 0
 5 lei are valoarea 1
10 lei are valoarea 2
20 lei are valoarea 3
50 lei are valoarea 4
100 lei are valoarea 5
500 lei are valoarea 6
1000 lei are valoarea 7
```

Remarcaţi faptul că bucla **for** este controlată de o variabilă de tipul **monede**. Deoarece o enumerare este un tip întreg, o valoare de tip enumerare poate fi utilizată oriunde în locul unei valori întregi. Observaţi însă că, la indexarea tabloului **nume**, este necesar un cast în faţa valorii de tip enumerare. O altă observaţie este că, deoarece valorile din enumerarea **monede** încep de la zero, ele se pot utiliza direct la indexarea tabloului **nume** pentru a obţine numele monedelor. Această metodă se dovedeşte utilă atunci când sunt necesare descrieri mai sugestive ale valorilor din enumerare.

Iniţializarea unei enumerări

Puteţi preciza în mod explicit valoarea uneia sau mai multor constante simbolice utilizând un iniţializator. Aceasta se obţine punând după constanta simbolică semnul egal şi o valoare întreagă. Constantele simbolice care urmează după cele iniţializate explicit vor primi valori mai mari decât cele situate înaintea lor. De exemplu, instrucţiunea următoare atribuie valoarea 100 constantei **oSutaLei**:

```
enum monede { unLeu, cinciLei, zeceLei, douazeciLei, cincizeciLei,
              oSutaLei = 100, cinciSuteLei, oMieLei };
```

Valorile constantelor simbolice sunt acum:

```
unLeu          0
cinciLei       1
zeceLei        2
douazeciLei    3
cincizeciLei   4
oSutaLei       100
cinciSuteLei   101
oMieLei        102
```

Precizarea tipului de bază al unei enumerări

În mod implicit, enumerările au ca tip de bază tipul **int**. Puteţi însă crea o enumerare de orice tip întreg, cu excepţia lui **char**. Pentru a specifica un alt tip de bază decât **int**, tipul trebuie precizat după numele enumerării, fiind separat de acesta prin două puncte. De exemplu, această instrucţiune face ca enumerarea **monede** să aibă tipul de bază **byte**:

```
enum monede : byte { unLeu, cinciLei, zeceLei, douazeciLei,
              cincizeciLei, cinciSuteLei, oMieLei };
```

Acum, **monede.zeceLei**, de exemplu, este o valoare **byte**.

Exerciţii la minut

- Este obligatoriu ca o structură să fie creată utilizând **new**?
- Pot structurile defini destructori?
- Ce este o enumerare?

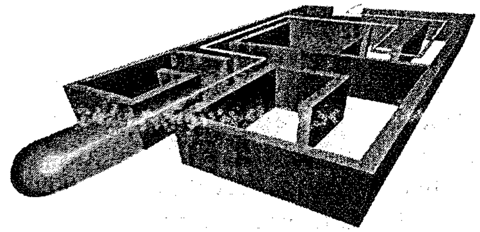
- Nu, un obiect **struct** poate fi creat la fel ca orice alt tip valoric, fără a utiliza **new**. În acest caz, însă, obiectul va fi neiniţializat.
- Nu. Structurile nu pot defini destructori.
- O enumerare este o listă de constante întregi care au nume.

Verificarea cunoştinţelor

1. „O singură interfaţă, mai multe metode" este un principiu de bază în C#. Care este facilitatea care ilustrează cel mai bine acest principiu?
2. Câte clase pot implementa o interfaţă? Câte interfeţe poate implementa o clasă dată?
3. Pot fi interfeţele moştenite?
4. Este obligatoriu ca o clasă să implementeze toţi membrii unei interfeţe?
5. Este posibil ca o interfaţă să declare un constructor?
6. Creaţi o interfaţă pentru clasa `Vehicle` din modulul 7. Denumiţi interfaţa `IVehicle`.
7. Creaţi o interfaţă pentru tablourile tolerante la erori. Pentru aceasta, adaptaţi exemplul cu tablouri tolerante la erori din modulul 7.
8. Ce diferenţă există între `struct` şi `class`?
9. Arătaţi cum puteţi crea o enumerare a planetelor din Sistemul Solar, în ordinea depărtării acestora faţă de Soare. Denumiţi această enumerare `Planete`.

Modulul 10

Tratarea excepţiilor



Scopurile acestui modul

- Utilizarea blocurilor `try` şi `catch`
- Efectele excepţiilor netratate
- Utilizarea instrucţiunilor `catch` multiple
- Imbricarea blocurilor `try`
- Generarea unei excepţii
- Studiul clasei `Exception`
- Utilizarea lui `finally`
- Studiul excepţiilor predefinite în C#
- Crearea claselor care definesc excepţii proprii
- Utilizarea lui `checked` şi `unchecked`

Acest modul prezintă tratarea excepţiilor. O *excepţie* este o eroare care se produce la momentul execuţiei. Utilizând sistemul de tratare a excepţiilor din C# puteţi trata, într-o manieră structurată şi controlată, erorile de execuţie. Abordarea propusă de C# pentru subiectul tratării excepţiilor păstrează similitudinea, aducând totodată îmbunătăţiri, cu metodele utilizate în C++ şi Java. Subiectul este desigur un teren cunoscut pentru cititorii cu experienţă în oricare din aceste limbaje. Sistemul de tratare a excepţiilor din C# se distinge însă printr-o implementare clară, uşor de utilizat.

Un avantaj esenţial al tratării excepţiilor este automatizarea într-o mare măsură a elaborării codului de tratare a erorilor, cod care anterior trebuia să fie introdus „de mână" în orice program de anvergură cât de cât mai mare. De exemplu, într-un limbaj de programare care nu tratează excepţii, toate metodele trebuie să întoarcă un cod de eroare atunci când eşuează, iar aceste coduri trebuie verificate manual la fiecare apel al metodei. Această abordare este greoaie şi predispusă la erori. Tratarea excepţiilor accelerează rezolvarea situaţilor de eroare, permiţând programului să definească un bloc de instrucţiuni, numit *rutină de tratare a excepţiilor*, care se execută în mod automat atunci când se produce o eroare. Nu mai este nevoie să verificaţi manual succesul sau eşecul fiecărei operaţii şi fiecărui apel de metodă în parte. Dacă se produce o eroare, aceasta va fi prelucrată în cadrul rutinei de tratare.

Un alt motiv pentru care tratarea excepţiilor este foarte importantă este că limbajul C# defineşte excepţii standard pentru tipurile obişnuite de erori dintr-un program,

cum ar fi împărţirea la zero sau indexarea unui tablou în afara marginilor. Pentru a răspunde la aceste erori, programul dumneavoastră trebuie să urmărească apariţia lor şi să le asigure tratarea.

În concluzie, rezultă că pentru a fi un programator de succes în C#, trebuie să stăpâniţi foarte bine sistemul de tratare a excepţiilor din cadrul limbajului.

Clasa `System.Exception`

În C#, excepţiile sunt reprezentate prin clase. Toate clasele care reprezintă excepţii trebuie să derive din clasa predefinită **`Exception`**, care face parte din spaţiul de nume **`System`**. Toate excepţiile sunt deci clase derivate din **`Exception`**.

Din clasa **`Exception`** derivă alte două clase, **`SystemException`** şi **`ApplicationException`**. Acestea implementează cele două categorii generale de excepţii definite de limbajul C#: cele generate de motorul de execuţie şi cele generate de programele de aplicaţie. Nici **`SystemException`** şi nici **`ApplicationException`** nu adaugă noi membri la clasa de bază **`Exception`**. Cele două clase definesc pur şi simplu rădăcinile celor două ierarhii de clase care reprezintă excepţii.

Limbajul C# defineşte câteva tipuri de excepţii care sunt derivate din **`SystemException`**. De exemplu, dacă se încearcă efectuarea unei împărţiri la zero, se produce o excepţie de tipul **`DivideByZeroException`**. După cum veţi învăţa ulterior în cadrul acestui modul, puteţi defini propriile clase care reprezintă excepţii, derivându-le din **`ApplicationException`**.

Noţiuni de bază despre tratarea excepţiilor

Tratarea excepţiilor în C# se realizează utilizând patru cuvinte cheie: **`try`**, **`catch`**, **`throw`** şi **`finally`**. Acestea definesc un sistem strâns conectat, utilizarea unuia dintre ele presupunând utilizarea altuia. Pe parcursul acestui modul, vom examina fiecare dintre aceste cuvinte cheie în detaliu. Pentru început, este bine să avem o idee generală despre rolul pe care fiecare dintre aceste construcţii îl are în tratarea excepţiilor. Pe scurt, iată cum funcţionează acestea.

Instrucţiunile din program care trebuie verificate pentru apariţia excepţiilor sunt incluse în corpul blocurilor **`try`** (încercare). Dacă pe parcursul execuţiei unui bloc **`try`** se produce o excepţie, aceasta este *lansată* (throw). Programul dumneavoastră poate intercepta această excepţie utilizând **`catch`**, tratând-o ulterior în conformitate cu logica situaţiei. Excepţiile de sistem sunt lansate în mod automat de către motorul de execuţie din C#. Pentru a lansa manual o excepţie, puteţi utiliza cuvântul cheie **`throw`**. Codul care trebuie neapărat executat la ieşirea dintr-un bloc **`try`** trebuie inclus într-un bloc **`finally`** (de final).

Utilizarea lui `try` şi `catch`

La baza tratării excepţiilor stau construcţiile **`try`** şi **`catch`**. Aceste cuvinte cheie funcţionează în pereche; nu este posibil să apară **`try`** fără **`catch`**, sau, invers, **`catch`** fără **`try`**. Forma generală a blocurilor **`try/catch`** utilizate la tratarea excepţiilor este cea de mai jos:

```
try {
    // bloc de cod monitorizat pt. detectarea erorilor
}

catch(TipExceptie1 exOb) {
    // rutina de tratare pt. TipExceptie1
}

catch(TipExceptie2 exOb) {
    // rutina de tratare pt. TipExceptie2
}
```

Mai sus, *TipExceptie* reprezintă tipul excepţiei care s-a produs. După ce este lansată, o excepţie este interceptată de instrucţiunea **`catch`** corespunzătoare, care prelucrează apoi excepţia. Cum se observă analizând forma generală de mai sus, pot exista mai multe instrucţiuni **`catch`** asociate unui bloc **`try`**. Tipul excepţiei stabileşte care din instrucţiunile **`catch`** se execută. Aşadar, dacă tipul excepţiei specificate de o instrucţiune **`catch`** coincide cu tipul excepţiei lansate, se execută acea instrucţiune **`catch`** (toate celelalte fiind ignorate). Execuţia continuă apoi cu instrucţiunea care urmează după ultimul **`catch`**. Instrucţiunile **`catch`** se execută deci numai dacă se lansează o excepţie. La interceptarea unei excepţii, valoarea acesteia este memorată de *exOb*.

Specificarea lui *exOb* este de fapt opţională. Dacă rutina de tratare nu are nevoie de acces la obiectul care reprezintă excepţia (situaţie frecvent întâlnită), nu este cazul să specificaţi *exOb*. Din acest motiv, multe din exemplele din acest modul nu vor specifica *exOb*.

Încă un aspect important: dacă nu se lansează nici o excepţie, blocul **`try`** se termină normal şi toate instrucţiunile **`catch`** sunt ignorate. Execuţia continuă şi în acest caz cu instrucţiunea care urmează după ultimul **`catch`**. Instrucţiunile **`catch`** se execută aşadar numai dacă s-a lansat o excepţie.

Primul exemplu simplu cu excepţii

În continuare, prezentăm un exemplu simplu care demonstrează cum puteţi aştepta producerea excepţiilor şi cum le puteţi apoi intercepta. După cum bine ştiţi, încercarea de indexare a unui tablou în afara marginilor sale reprezintă o eroare. În acest caz, motorul de execuţie din C# lansează o excepţie de tipul **`IndexOutOfRangeException`**, unul din tipurile predefinite în C#. Programul următor generează în mod intenţionat o astfel de excepţie, pe care mai apoi o interceptează:

```
// Prezinta tratarea exceptiilor.
using System;

class ExcDemo1 {
  public static void Main() {
    int [] nums = new int[4];

    try {
      Console.WriteLine("Inainte de generarea exceptiei.");

      // Generam o exceptie de depasire a marginilor.
      nums[7] = 10; <-- Încercare de indexare în afara marginilor.
      Console.WriteLine("Mesaj care nu se afiseaza");
    }
    catch(IndexOutOfRangeException) {
      // interceptam exceptia
      Console.WriteLine("Indexul depaseste marginile!");
    }
    Console.WriteLine("Dupa instructiunea catch.");
  }
}
```

Acest program afişează următorul text:

```
Inainte de generarea exceptiei.
Indexul depaseste marginile!
Dupa instructiunea catch.
```

Deşi este destul de scurt, programul anterior prezintă câteva aspecte esenţiale legate de tratarea excepţiilor. Mai întâi, codul care trebuie monitorizat pentru detectarea erorilor este conţinut în interiorul unui bloc **try**. În al doilea rând, atunci când se produce o excepţie (determinată, în acest caz, de încercarea de a indexa tabloul **nums** dincolo de marginile sale), aceasta este lansată de blocul **try** şi interceptată de instrucţiunea **catch**. În acest moment, controlul programului este transferat instrucţiunii **catch**, iar blocul **try** se termină. Instrucţiunea **catch** nu este aşadar apelată, ci controlul execuţiei programului i se transferă imediat. Instrucţiunea **WriteLine()** situată după accesul la indexul care depăşeşte marginile nu se execută niciodată. După executarea instrucţiunii **catch**, controlul programului se transferă instrucţiunilor care urmează după **catch**. Remedierea problemei care a produs excepţia, astfel încât execuţia programului să poată continua în mod normal, este misiunea rutinei de tratare a excepţiei.

Remarcaţi că instrucţiunea **catch** nu specifică nici un parametru. După cum am menţionat, parametrul este necesar numai dacă aveţi nevoie de accesul la obiectul care reprezintă excepţia. În unele cazuri, valoarea acestui obiect poate fi utilizată de rutina de tratare a excepţiei pentru obţinerea unor informaţii suplimentare legate de eroarea produsă; în majoritatea cazurilor însă, este suficient să ştiţi că excepţia s-a produs. Lipsa parametrului instrucţiunii **catch** din rutina de tratare (lipsă remarcată şi în exemplul anterior) nu constituie deci un fapt neobişnuit.

După cum am precizat, dacă un bloc **try** nu lansează nici o excepţie, nu se va executa nici o instrucţiune **catch**, iar controlul programului se transferă după ultima instrucţiune **catch**. Pentru a vă convinge de aceasta, înlocuiţi linia:

```
nums[7] = 10;
```

cu:

```
nums[0] = 10;
```

În acest caz, nu se mai generează nici o excepţie, iar blocul **catch** nu se execută.

Al doilea exemplu cu excepţii

Este foarte important să înţelegeţi că întreg codul dintr-un bloc **try** este monitorizat pentru detectarea excepţiilor. Monitorizarea se referă şi la excepţiile pe care le poate genera o metodă care este apelată din interiorul blocului **try**. O excepţie lansată de o metodă apelată din interiorul unui bloc **try** poate fi interceptată de acel bloc **try** – presupunând, desigur, că metoda nu interceptează ea însăşi excepţia. De exemplu, programul de mai jos este corect:

```
/* O exceptie poate fi generata de o metoda si
   interceptata de alta metoda. */
using System;

class ExcTest {
  // Generam o exceptie.
  public static void genException() {
    int [] nums = new int[4];

    Console.WriteLine("Inainte de generarea exceptiei.");

    // Generam o exceptie de depasire a marginilor.
    nums[7] = 10; <-- Se generează o excepţie
    Console.WriteLine("Mesaj care nu se afiseaza");
  }
}

class ExcDemo2 {
  public static void Main() {

    try {
      ExcTest.genException();
    }
    catch(IndexOutOfRangeException) { <-- Se interceptează excepţia.
      // interceptam exceptia
      Console.WriteLine("Indexul depaseste marginile!");
    }
    Console.WriteLine("Dupa instructiunea catch.");
  }
}
```

Acest program afişează următorul text, identic cu cel afişat de prima versiune, prezentată anterior:

```
Inainte de generarea exceptiei.
Indexul depaseste marginile!
Dupa instructiunea catch.
```

Deoarece `genException()` este apelată dintr-un bloc `try`, excepția pe care o generează (fără să o intercepteze) este interceptată de instrucțiunea `catch` din `Main()`. Rețineți, însă, că dacă metoda `genException()` ar fi interceptat ea însăși excepția, controlul nu ar mai fi revenit niciodată înapoi în `Main()`.

Exerciții la minut

- Ce este o excepție?
- Ce bloc include codul monitorizat pentru detectarea excepțiilor?
- Care este rolul lui `catch`? După executarea lui `catch`, ce se întâmplă cu fluxul execuției programului?

Consecințele excepțiilor neinterceptate

Interceptarea uneia dintre excepțiile standard din C#, prezentată și în programul anterior, are și un avantaj secundar: evită terminarea anormală a programului. Atunci când o excepție este lansată, ea trebuie interceptată de o secvență de cod din program. În general, dacă programul dumneavoastră nu interceptează excepția, aceasta va fi interceptată de motorul de execuție din C#. Dezavantajul este că motorul de execuție va raporta apariția unei erori și va forța terminarea programului. De exemplu, în această versiune a exemplului precedent, excepția de depășire a marginilor nu mai este interceptată de program:

```
// Lasam tratarea erorii in sarcina motorului de executie din C#.
using System;

class NotHandled {
  public static void Main() {
    int [] nums = new int[4];

    Console.WriteLine("Inainte de generarea exceptiei.");

    // Generam o exceptie de depasire a marginilor.
    nums[7] = 10;
  }
}
```

- O excepție este o eroare la execuție.
- Pentru a fi monitorizat în vederea detectării excepțiilor, codul trebuie să facă parte dintr-un bloc **try**.
- Instrucțiunea **catch** interceptează excepții. O instrucțiune **catch** nu este apelată, deci execuția *nu revine* în locul în care a fost generată excepția. Execuția va continua cu instrucțiunile situate după blocul **catch**.

Atunci când se produce excepția de depășire a marginii tabloului, execuția programului se oprește, afișând următorul mesaj de eroare:

```
Unhandled Exception: System.IndexOutOfRangeException:
        Exception of type System.IndexOutOfRangeException
        was thrown.
at NotHandled.Main()
```

Chiar dacă un astfel de mesaj vă este util la depanarea programului, acesta nu reprezintă în nici un caz ceea ce trebuie să vadă utilizatorii. De aceea, este foarte important ca programul dumneavoastră să se ocupe singur de tratarea excepțiilor.

După cum am precizat mai devreme, tipul excepției trebuie să coincidă cu tipul specificat într-o instrucțiune `catch`. În caz contrar, excepția nu va fi interceptată. De exemplu, programul următor încearcă să intercepteze o eroare de depășire a marginii unui tablou cu o instrucțiune `catch` corespunzătoare tipului `DivideByZeroException` (un alt tip predefinit de excepții în C#). La depășirea marginii tabloului, se va lansa o excepție de tipul `IndexOutOfRangeException`, dar aceasta nu este interceptată de instrucțiunea `catch`. În consecință, se va produce terminarea forțată a programului.

```
// nu functioneaza !
using System;

class ExcTypeMismatch {
  public static void Main() {
    int [] nums = new int[4];

    try {
      Console.WriteLine("Inainte de generarea exceptiei.");

      // Generam o exceptie de depasire a marginilor.
      nums[7] = 10;
      Console.WriteLine("Mesaj care nu se afiseaza");
    }

    /* nu putem intercepta eroarea de depasire a marginii
       specificand tipul DivideByZeroException. */
    catch(DivideByZeroException) {
      // interceptam exceptia
      Console.WriteLine("Indexul depaseste marginile!");
    }
    Console.WriteLine("Dupa instructiunea catch.");
  }
}
```

Lansează o excepție **IndexOutOfRangeException**.

Încearcă să intercepteze **DivideByZeroException**.

Rezultatul programului este prezentat mai jos:

```
Inainte de generarea exceptiei.

Unhandled Exception: System.IndexOutOfRangeException:
        Exception of type System.IndexOutOfRangeException
        was thrown.
at ExcTypeMismatch.Main()
```

După cum arată acest rezultat, o instrucțiune **catch** pentru tipul **DivideByZeroException** nu poate intercepta o excepție **IndexOutOfRangeException**.

Excepțiile permit tratarea elegantă a erorilor

Unul dintre avantajele esențiale ale tratării excepțiilor este acela că permite programului dumneavoastră să răspundă unei situații de eroare, continuându-și apoi execuția. De exemplu, să considerăm următorul exemplu, care împarte elementele unui tablou la elementele din alt tablou. Atunci când se produce o împărțire la zero, se generează o excepție de tipul **DivideByZeroException**. În program, această excepție este tratată prin raportarea erorii, după care execuția se reia. Încercarea de împărțire la zero nu produce deci o eroare abruptă de execuție, care are ca rezultat terminarea imediată a programului. Eroarea este tratată în mod elegant, iar execuția programului poate continua.

```
// Tratarea eleganta a erorii si continuarea executiei programului.
using System;

class ExcDemo3 {
  public static void Main() {
    int [] numarator = { 4, 8, 16, 32, 64, 128 };
    int [] numitor = { 2, 0, 4, 4, 0, 8 };

    for(int i = 0; i < numarator.Length; i++) {
      try {
        Console.WriteLine(numarator[i] + " / " +
                          numitor[i] + " este " +
                          numarator[i]/numitor[i]);
      }
      catch(DivideByZeroException) {
        // interceptam exceptia
        Console.WriteLine("Nu putem imparti la zero !");
      }
    }
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
4 / 2 este 2
Nu putem imparti la zero !
16 / 4 este 4
32 / 4 este 8
Nu putem imparti la zero !
128 / 8 este 16
```

Acest exemplu evidențiază și un alt aspect: după ce o excepție a fost tratată, este eliminată din sistem. În consecință, în program, de fiecare dată când bucla **for** intră din nou în blocul **try**, este sigur că toate excepțiile apărute anterior au fost tratate. Aceasta permite programului să trateze erorile care apar în mod repetat.

Exerciții la minut

- Are vreo importanță tipul excepției dintr-o instrucțiune **catch**?
- Ce se întâmplă atunci când o excepție nu este interceptată?
- Ce trebuie să facă programul dumneavoastră atunci când se produce o excepție?

Utilizarea instrucțiunilor catch multiple

Puteți asocia unui bloc **try** mai multe instrucțiuni **catch**. De fapt, așa veți și proceda în cele mai multe cazuri. Fiecare instrucțiune trebuie însă să intercepteze un tip diferit de excepții. De exemplu, programul prezentat în continuare interceptează atât erori de depășire a marginii unui tablou, cât și erori de împărțire la zero.

```
// Utilizarea instructiunilor catch multiple.
using System;

class ExcDemo4 {
  public static void Main() {
    // Tabloul numarator este mai lung decat numitor.
    int [] numarator = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int [] numitor = { 2, 0, 4, 4, 0, 8 };

    for(int i = 0; i < numarator.Length; i++) {
      try {
        Console.WriteLine(numarator[i] + " / " +
                          numitor[i] + " este " +
                          numarator[i]/numitor[i]);
      }
      catch(DivideByZeroException) {
        // interceptam exceptia
        Console.WriteLine("Nu putem imparti la zero !");
      }
      catch(IndexOutOfRangeException) {
        // interceptam exceptia
        Console.WriteLine("Elementul nu exista in tablou.");
      }
    }
  }
}
```

Instrucțiuni **catch** multiple.

Programul va afișa următorul rezultat:

```
4 / 2 este 2
Nu putem imparti la zero !
16 / 4 este 4
```

- Tipul excepției dintr-o instrucțiune **catch** trebuie să coincidă cu tipul excepției pe care doriți să o interceptați.
- O excepție care nu este interceptată conduce în final la terminarea forțată a programului.
- Un program trebuie să trateze excepțiile într-o manieră logică și elegantă, eliminând cauza producerii lor dacă este posibil și continuându-și apoi execuția.

```
32 / 4 este 8
Nu putem imparti la zero !
128 / 8 este 16
Elementul nu exista in tablou.
Elementul nu exista in tablou.
```

După cum confirmă acest rezultat, fiecare instrucțiune **catch** corespunde numai cu propriul tip de excepție.

În general, expresiile **catch** sunt verificate în ordinea apariției lor în program. Numai o singură instrucțiune care se potrivește este executată. Toate celelalte blocuri **catch** sunt ignorate.

Interceptarea tuturor excepțiilor

Uneori, aveți nevoie să interceptați toate excepțiile, indiferent de tipul lor. Pentru aceasta, utilizați o instrucțiune **catch** fără nici un parametru. Aceasta creează o rutină „universală", care este utilă dacă doriți să vă asigurați că programul dumneavoastră tratează toate excepțiile. În exemplul următor, singura instrucțiune **catch** este cea universală, care interceptează atât excepțiile **IndexOutOfRangeException**, cât și pe cele **DivideByZeroException**, generate de program:

```
// Utilizarea unei instructiuni catch universale.
using System;

class ExcDemo5 {
  public static void Main() {
    // Tabloul numarator este mai lung decat numitor.
    int [] numarator = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int [] numitor = { 2, 0, 4, 4, 0, 8 };

    for(int i = 0; i < numarator.Length; i++) {
      try {
        Console.WriteLine(numarator[i] + " / " +
                          numitor[i] + " este " +
                          numarator[i]/numitor[i]);
      }
      catch {   <-- Interceptează toate excepțiile.
        Console.WriteLine("S-a produs o exceptie.");
      }
    }
  }
}
```

Rezultatul este prezentat mai jos:

```
4 / 2 este 2
S-a produs o exceptie.
16 / 4 este 4
32 / 4 este 8
S-a produs o exceptie.
128 / 8 este 16
S-a produs o exceptie.
S-a produs o exceptie.
```


Blocurile try pot fi imbricate

Este posibil ca un bloc **try** să fie conținut în interiorul altuia. Excepțiile generate de blocul **try** interior și care nu sunt interceptate de instrucțiunile **catch** asociate acelui bloc sunt transmise blocului **try** exterior. În exemplul de mai jos, excepția **IndexOutOfRangeException** nu este interceptată de blocul **try** interior, ci de cel exterior.

```
// Utilizarea blocurilor try imbricate.
using System;

class NestTrys {
  public static void Main() {
    // Tabloul numarator este mai lung decat numitor.
    int [] numarator = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int [] numitor = { 2, 0, 4, 4, 0, 8 };

    try { // try exterior   <-- Blocuri try imbricate.
      for(int i = 0; i < numarator.Length; i++) {
        try {  // try imbricat   <--
          Console.WriteLine(numarator[i] + " / " +
                            numitor[i] + " este " +
                            numarator[i]/numitor[i]);
        }
        catch(DivideByZeroException) {
          // interceptam exceptia
          Console.WriteLine("Nu putem imparti la zero !");
        }
      }
    }
    catch(IndexOutOfRangeException) {
      // interceptam exceptia
      Console.WriteLine("Elementul nu exista in tablou.");
      Console.WriteLine("Eroare critica - programul se termina.");
    }
  }
}
```

Rezultatul programului este prezentat mai jos:

```
4 / 2 este 2
Nu putem imparti la zero !
16 / 4 este 4
32 / 4 este 8
Nu putem imparti la zero !
128 / 8 este 16
Elementul nu exista in tablou.
Eroare critica - programul se termina.
```

În acest exemplu, excepțiile tratate de blocul **try** interior – în acest caz, erorile de împărțire la zero – permit continuarea execuției programului. Eroarea de depășire a marginii tabloului este însă interceptată de blocul **try** exterior, conducând așadar la terminarea programului.

Deși acesta nu este singurul motiv pentru care puteți utiliza blocuri **try** imbricate, programul anterior prezintă un aspect important, care poate fi generalizat. Adeseori,

blocurile **try** imbricate se utilizează pentru a permite tratarea diferențiată a categoriilor diferite de erori. Unele tipuri de erori sunt critice și nu pot fi remediate. Altele sunt minore și pot fi tratate imediat. Mulți programatori utilizează un bloc **try** exterior pentru a intercepta erorile cele mai severe, în timp ce erorile mai puțin grave sunt tratate de blocuri **try** interioare. Puteți de asemenea să utilizați un bloc **try** exterior ca „bloc universal" pentru interceptarea erorilor care nu sunt tratate de blocurile interioare.

Exerciții la minut

- Cum puteți intercepta toate excepțiile?
- Puteți utiliza același bloc **try** pentru a intercepta două sau mai multe tipuri diferite de excepții?
- În cazul blocurilor **try** imbricate, ce se întâmplă cu o excepție care nu este interceptată de blocul interior?

Lansarea unei excepții

Exemplele precedente interceptează excepții generate automat de C#. Este însă posibil să lansați manual o excepție, utilizând o instrucțiune **throw**. Forma generală a acesteia este prezentată mai jos:

throw *exceptOb;*

Mai sus, *exceptOb* trebuie să fie un obiect al cărui tip trebuie să fie o clasă derivată din **Exception**.

În continuare, prezentăm un exemplu care utilizează instrucțiunea **throw** pentru a lansa o excepție **DivideByZeroException**:

```
// Lansarea manuala a unei exceptii.
using System;

class ThrowDemo {
  public static void Main() {
    try {
      Console.WriteLine("Inainte de throw.");
      throw new DivideByZeroException();   ◄── Lansarea unei excepții.
    }
    catch(DivideByZeroException) {
      // interceptam exceptia
      Console.WriteLine("Exceptie interceptata.");
    }
```

- Pentru a intercepta toate excepțiile, utilizați o instrucțiune **catch** fără parametri.
- Da, utilizați câte o instrucțiune **catch** pentru fiecare excepție pe care doriți să o interceptați.
- O excepție care nu este interceptată de un bloc **try/catch** interior migrează spre blocul **try** exterior.


```
    Console.WriteLine("Dupa blocul try/catch.");
  }
}
```

Rezultatul programului este prezentat mai jos:

```
Inainte de throw.
Exceptie interceptata.
Dupa blocul try/catch.
```

Remarcați modul în care am creat obiectul **DivideByZeroException**, utilizând **new** în instrucțiunea **throw**. Amintiți-vă că instrucțiunea **throw** lansează un obiect. În concluzie, trebuie ca obiectul ce va fi lansat să fie mai întâi creat. Cu alte cuvinte, nu puteți lansa un tip generic de excepție. În acest caz, s-a utilizat constructorul implicit pentru crearea unui obiect **DivideByZeroException**, dar excepțiile dispun și de alți constructori (pe care îi veți studia ulterior).

Sfatul expertului

Întrebare: În ce caz este necesară lansarea manuală a unei excepții?

Răspuns: În majoritatea cazurilor, excepțiile pe care le veți lansa vor fi instanțe ale claselor reprezentând excepții create chiar de dumneavoastră. După cum veți vedea ulterior în acest modul, crearea propriilor clase care definesc excepții vă permite să tratați erorile specifice generate de codul dumneavoastră utilizând o strategie globală de tratare pe întreg parcursul programului.

Relansarea unei excepții

O excepție interceptată de o instrucțiune **catch** poate fi relansată, cu scopul de a fi interceptată de un bloc **catch** exterior. În cele mai multe cazuri, excepțiile sunt relansate pentru a putea fi prelucrate de mai multe rutine de tratare. De exemplu, o primă rutină poate trata un aspect al excepției, lăsând celelalte aspecte în sarcina unei alte rutine. Pentru relansarea unei excepții, se utilizează forma simplă a instrucțiunii **throw**, fără a specifica nici o excepție. Veți utiliza așadar următoarea formă a lui **throw**:

throw;

Rețineți că atunci când relansați o excepție, ea nu va fi interceptată de aceeași instrucțiune **catch**. Excepția se va propaga la următoarea instrucțiune **catch**.

Programul următor prezintă modul în care puteți relansa o excepție:

```
// Relansarea unei exceptii.
using System;

class Rethrow {
  public static void genException() {
    // Tabloul numarator este mai lung decat numitor.
    int [] numarator = { 4, 8, 16, 32, 64, 128, 256, 512 };
```

```
    int [] numitor = { 2, 0, 4, 4, 0, 8 };

    for(int i = 0; i < numarator.Length; i++) {
      try {
        Console.WriteLine(numarator[i] + " / " +
                          numitor[i] + " este " +
                          numarator[i]/numitor[i]);
      }
      catch(DivideByZeroException) {
        // interceptam exceptia
        Console.WriteLine("Nu putem imparti la zero !");
      }
      catch(IndexOutOfRangeException) {
        // interceptam exceptia
        Console.WriteLine("Elementul nu exista in tablou.");
        throw; // relansam exceptia
      }
    }
  }
}

class RethrowDemo {
  public static void Main() {
    try {
      Rethrow.genException();
    }
    catch(IndexOutOfRangeException) {
      // interceptam din nou exceptia
      Console.WriteLine("Eroare critica - programul se termina.");
    }
  }
}
```

Relansam excepția.

Interceptăm excepția relansată.

În acest program, erorile de împărțire la zero sunt tratate local, în metoda **genException()**, dar erorile de depășire a marginii tabloului sunt relansate. În acest caz, ele sunt interceptate de **Main()**.

Exerciții la minut

- Ce rol are **throw**?
- Instrucțiunea **throw** lucrează cu tipuri de excepții sau cu obiecte?
- Poate fi o excepție relansată după ce a fost interceptată? În caz afirmativ, ce formă a lui **throw** veți utiliza?

- Instrucțiunea **throw** generează o excepție.
- **throw** lucrează cu obiecte. Acestea trebuie să fie instanțe ale unor clase care reprezintă excepții, desigur.
- Da. Se utilizează **throw** fără a specifica nici o excepție.

Utilizarea lui finally

În unele cazuri, este necesar să definiți un bloc de instrucțiuni care să se execute la terminarea unui bloc **try/catch**. De exemplu, o excepție poate corespunde unei erori care cauzează terminarea prematură a metodei curente. Este însă posibil ca metoda să fi deschis un fișier sau o conexiune în rețea care trebuie închisă. Astfel de situații se întâlnesc frecvent în programare, iar limbajul C# oferă o modalitate convenabilă de tratare a lor: **finally**.

Pentru a specifica un bloc de cod care să se execute la ieșirea dintr-un bloc **try/catch**, includeți un bloc **finally** la sfârșitul secvenței **try/catch**. Forma generală a unui bloc **try/catch** care include **finally** este prezentată mai jos:

```
try {
   // bloc de cod monitorizat pt. detectarea erorilor
}

catch(TipExceptie1 exOb) {
   // rutina de tratare pt. TipExceptie1
}

catch(TipExceptie2 exOb) {
   // rutina de tratare pt. TipExceptie2
}

finally {
   // bloc final
}
```

Blocul **finally** se execută după ce execuția unui bloc **try/catch** se termină, indiferent de condiția care a produs terminarea. Cu alte cuvinte, chiar dacă blocul **try** se termină normal sau din cauza unei excepții, ultima secvență de cod care se execută este cea definită de **finally**. Blocul **finally** se va executa și atunci când blocul **try** sau oricare din instrucțiunile **catch** asociate determină revenirea din metodă.

Mai jos, prezentăm un exemplu care utilizează **finally**:

```
// Utilizarea lui finally.
using System;

class UseFinally {
  public static void genException(int what) {
    int t;
    int [] nums = new int[2];

    Console.WriteLine("Parametrul " + what);
    try {
      switch(what) {
        case 0:
          t = 10 / what; // eroare de impartire la zero
          break;
```

```
        case 1:
          nums[4] = 4;    // depasire a marginii
          break;
        case 2:
          return;    // revenire din blocul try
      }
    }
    catch(DivideByZeroException) {
      // interceptam exceptia
      Console.WriteLine("Nu se poate imparti la zero!");
      return;    // revenire din catch
    }
    catch(IndexOutOfRangeException) {
      // interceptam exceptia
      Console.WriteLine("Elementul nu exista in tablou.");
    }
    finally {  <-- Se execută la terminarea blocului try/catch.
      Console.WriteLine("Iesire din blocul try.");
    }
  }
}

class FinallyDemo {
  public static void Main() {

    for(int i = 0; i < 3; i++) {
      UseFinally.genException(i);
      Console.WriteLine();
    }
  }
}
```

Rezultatul produs de program este prezentat mai jos:

```
Parametrul 0
Nu se poate imparti la zero!
Iesire din blocul try.

Parametrul 1
Elementul nu exista in tablou.
Iesire din blocul try.

Parametrul 2
Iesire din blocul try.
```

După cum arată rezultatul, blocul **finally** se execută indiferent de modul în care se iese din blocul **try**.

Studiul detaliat al clasei Exception

Până în prezent, am interceptat excepţiile, fără însă a utiliza obiectele prin care acestea sunt reprezentate. După cum am precizat anterior, instrucţiunile **catch** vă permit să specificaţi un tip de excepţie *şi* un parametru. Parametrul este obiectul care reprezintă excepţia. Deoarece toate excepţiile sunt derivate din clasa **Exception**,

ele vor permite accesul la membrii definiţi de clasa **Exception**. În continuare, vom studia câţiva dintre membrii şi constructorii cei mai utili ai acestei clase şi vom începe să utilizăm parametrul instrucţiunilor **catch**.

Clasa **Exception** defineşte câteva proprietăţi. Trei dintre cele mai interesante proprietăţi sunt **Message**, **StackTrace** şi **TargetSite**. Toate trei sunt accesibile numai la citire. **Message** conţine un string care descrie natura erorii. **StackTrace** conţine un alt string, care prezintă stiva apelurilor care a condus la apariţia excepţiei. **TargetSite** întoarce un obiect care specifică metoda care a generat excepţia.

Clasa **Exception** defineşte şi câteva metode. Cea pe care o veţi utiliza cel mai frecvent este **ToString()**, care întoarce un **string** care descrie excepţia. Metoda **ToString()** este apelată automat atunci când excepţia este afişată prin intermediul lui **WriteLine()**.

Programul următor utilizează proprietăţile şi metoda prezentate mai sus:

```
// Utilizarea membrilor clase Exception.
using System;

class ExcTest {
  public static void genException() {
    int [] nums = new int[4];

    Console.WriteLine("Inainte de generarea exceptiei.");

    // generam o exceptie de depasire a marginii
    nums[7] = 10;
    Console.WriteLine("Acest mesaj nu va fi afisat");
  }
}

class UseExcept {
  public static void Main() {

    try {
      ExcTest.genException();
    }
    catch(IndexOutOfRangeException exc) {
      // interceptam exceptia
      Console.WriteLine("Mesajul standard este: ");
      Console.WriteLine(exc);  // apel la ToString()
      Console.WriteLine("Stiva de apeluri: " + exc.StackTrace);
      Console.WriteLine("Mesajul: " + exc.Message);
      Console.WriteLine("Metoda care a generat exceptia: " + exc.TargetSite);
    }
    Console.WriteLine("Dupa instructiunea catch.");
  }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Inainte de generarea exceptiei.
Mesajul standard este:
System.IndexOutOfRangeException: Exception of type
                     System.IndexOutOfRangeException was thrown.
   at ExcTest.genException()
   at UseExcept.Main()
```

```
Stiva de apeluri: at ExcTest.genException()
     at UseExcept.Main()
Mesajul: Exception of type System.IndexOutOfRangeException
         was thrown.
Metoda care a generat exceptia: Void genException()
Dupa instructiunea catch.
```

Clasa **Exception** defineşte patru constructori. Cei doi pe care îi vom studia sunt prezentaţi mai jos:

Exception()

Exception(string *str***)**

Primul este constructorul implicit. Cel de-al doilea specifică valoarea proprietăţii **Message** care va fi asociată excepţiei. Atunci când creaţi propriile dumneavoastră clase care reprezintă excepţii, trebuie să implementaţi amândoi constructorii de mai sus.

Excepţii mai frecvent utilizate

Spaţiul de nume **System** defineşte câteva tipuri standard de excepţii. Toate aceste tipuri sunt derivate din **SystemException**, deoarece sunt generate de motorul de programare atunci când se produc erori la execuţia programului. Câteva dintre excepţiile standard definite în C#, cel mai frecvent utilizate, sunt prezentate în tabelul 10-1.

Tabelul 10-1 Excepţii frecvent utilizate definite în spaţiul de nume **System**.

Excepţia	Semnificaţia
ArrayTypeMismatchException	Tipul valorii memorate este incompatibil cu tipul tabloului.
DivideByZeroException	Încercare de împărţire la zero.
IndexOutOfRangeException	Indexul din tablou depăşeşte marginile.
InvalidCastException	Cast incorect la execuţie.
OutOfMemoryException	Apelul lui **new** eşuează din cauza memoriei insuficiente.
OverflowException	Depăşire aritmetică.
StackOverflowException	Depăşirea capacităţii stivei.

Exerciţii la minut

- Ce conţine proprietatea **Message**?
- Când se execută instrucţiunile din blocul **finally**?
- Cum puteţi afişa o imagine a stivei care prezintă succesiunea apelurilor care conduc la producerea unei excepţii?

- **Message** conţine un mesaj care descrie excepţia.
- Blocul **finally** se execută ultimul, după ieşirea dintr-un bloc **try**.
- Pentru afişarea unei imagini a stivei, afişaţi proprietatea **StackTrace** definită de clasa **Exception**.

Clase derivate din Exception

Deşi erorile cel mai frecvent întâlnite sunt tratate utilizând excepţiile predefinite în C#, mecanismul de tratare a excepţiilor din C# nu se limitează numai la acestea. De fapt, o parte din puterea mecanismului de excepţii din C# o constituie capacitatea de a trata excepţiile definite de utilizator. Puteţi utiliza excepţii personalizate pentru tratarea erorilor specifice care pot apărea în programul dumneavoastră. Crearea unui nou tip de excepţii este simplă. Este suficient să definiţi o clasă derivată din **Exception**. Ca o regulă generală, excepţiile definite de dumneavoastră trebuie să fie derivate din clasa **ApplicationException**, aceasta fiind rădăcina ierarhiei rezervate pentru excepţiile generate de aplicaţii. Clasele derivate de dumneavoastră nu trebuie de fapt să implementeze nimic – simpla lor apartenenţă la ierarhia de tipuri permite utilizarea lor ca excepţii.

Clasele create de dumneavoastră pentru a reprezenta excepţii dispun în mod automat de proprietăţile şi metodele definite de clasa **Exception**. Desigur, puteţi extinde unul sau mai mulţi dintre aceşti membri în clasele reprezentând excepţii pe care le veţi crea.

În continuare, iată un exemplu care creează o excepţie numită **NonIntResultException**. În program, excepţia este lansată atunci când împărţirea a două valori întregi produce un rezultat cu o parte fracţionară. Clasa **NonIntResultException** defineşte cei doi constructori standard şi redefineşte metoda **ToString()**.

```
// Utilizarea unei exceptii personalizate.
using System;

// Cream o exceptie.
class NonIntResultException : ApplicationException {
  // Implementarea constructorilor standard
  public NonIntResultException() : base() { }
  public NonIntResultException(string str) : base(str) { }

  // Extindem metoda ToString
  // pentru NonIntResultException.
  public override string ToString() {
    return Message;
  }
}

class CustomExceptDemo {
  public static void Main() {

    // Tabloul numarator contine aici si valori impare.
    int [] numarator = { 4, 8, 15, 32, 64, 127, 256, 512 };
    int [] numitor = { 2, 0, 4, 4, 0, 8 };

    for(int i = 0; i < numarator.Length; i++) {
      try {
        if((numarator[i] % 2) != 0)
```

O excepţie personalizată.

Extindem **ToString()**.

```
      throw new
        NonIntResultException("Rezultatul " +
            numarator[i] + " / " + numitor[i] + " nu este intreg.");

    Console.WriteLine(numarator[i] + " / " +
                      numitor[i] + " este " +
                      numarator[i]/numitor[i]);
  }
  catch(DivideByZeroException) {
    // interceptam exceptia
    Console.WriteLine("Nu putem imparti la zero !");
  }
  catch(IndexOutOfRangeException) {
    // interceptam exceptia
    Console.WriteLine("Elementul nu exista in tablou.");
  }
  catch(NonIntResultException exc) {
    Console.WriteLine(exc);
  }
 }
}
}
```

Lansăm o excepţie personalizată.

Rezultatul programului este prezentat mai jos:

```
4 / 2 este 2
Nu putem imparti la zero !
Rezultatul 15 / 4 nu este intreg.
32 / 4 este 8
Nu putem imparti la zero !
Rezultatul 127 / 8 nu este intreg.
Elementul nu exista in tablou.
Elementul nu exista in tablou.
```

Înainte de a trece mai departe, puteţi experimenta câteva modificări asupra acestui program. De exemplu, încercaţi să includeţi în comentariu versiunea extinsă a metodei **ToString()** şi observaţi rezultatul afişat. De asemenea, încercaţi să creaţi o excepţie utilizând constructorul implicit şi observaţi care este mesajul implicit generat de C#.

Interceptarea excepţiilor derivate

Trebuie să fiţi foarte atent la ordonarea instrucţiunilor **catch** atunci când încercaţi să interceptaţi excepţii ale căror tipuri sunt atât clase de bază, cât şi clase derivate, deoarece o instrucţiune **catch** pentru o clasă de bază va intercepta şi excepţiile corespunzătoare claselor derivate. De exemplu, deoarece clasa **Exception** este clasa de bază pentru toate excepţiile, interceptarea tipului **Exception** înseamnă de fapt interceptarea tuturor excepţiilor posibile. Desigur, o modalitate mai elegantă de a intercepta toate excepţiile este utilizarea lui **catch** fără parametri, prezentată anterior. Subiectul interceptării excepţiilor de tipuri derivate se dovedeşte însă foarte important în alte contexte, în special pentru excepţiile create de dumneavoastră.

Dacă trebuie să interceptaţi excepţiile având ca tipuri atât o clasă de bază, cât şi o clasă derivată, atunci clasa derivată trebuie să fie prima în secvenţa de instrucţiuni **catch**. În caz contrar, instrucţiunea **catch** asociată clasei de bază va intercepta şi excepţiile claselor derivate din ea. Această regulă se impune de la sine, deoarece apariţia clasei de bază înaintea clasei derivate conduce la existenţa codului inaccesibil, întrucât instrucţiunea **catch** asociată clasei derivate nu se va executa niciodată. În C#, prezenţa unei instrucţiuni **catch** inaccesibile este considerată eroare.

Programul următor creează două clase care reprezintă excepţii, denumite **ExceptA** şi **ExceptB**. Clasa **ExceptA** este derivată din **ApplicationException**, iar **ExceptB** este derivată din **ExceptA**. Programul lansează apoi câte o excepţie de fiecare tip.

```
// Exceptiile derivate trebuie sa apara inaintea celor din
// clasa de baza
using System;

// Cream un tip de exceptii.
class ExceptA : ApplicationException {
  public ExceptA() : base() { }
  public ExceptA(string str) : base(str) { }

  public override string ToString() {
    return Message;
  }
}

// Cream un tip de exceptii derivat din ExceptA
class ExceptB : ExceptA {
  public ExceptB() : base() { }
  public ExceptB(string str) : base(str) { }

  public override string ToString() {
    return Message;
  }
}

class OrderMatters {
  public static void Main() {
    for(int x = 0; x < 3; x++) {
      try {
        if(x == 0) throw new ExceptA("Exceptie de tipul ExceptA");
        else if(x == 1) throw new ExceptB("Exceptie de tipul ExceptB");
        else throw new Exception();
      }
      catch(ExceptB exc) {
        // interceptam exceptia
        Console.WriteLine(exc);
      }
      catch(ExceptA exc) {
        // interceptam exceptia
        Console.WriteLine(exc);
      }
      catch(Exception exc) {
        Console.WriteLine(exc);
      }
```

Se derivează un tip de excepţii.

Se derivează din **ExceptA** un alt tip de excepţii.

Ordinea instrucţiunilor **catch** contează.

```
    }
  }
}
```

Sfatul expertului

Întrebare: Din moment ce excepțiile indică de regulă erori bine definite, de ce este nevoie să interceptăm excepții având tipul unor clase de bază?

Răspuns: O clauză **catch** care interceptează o clasă de bază vă permite să tratați o întreagă categorie de excepții utilizând o singură instrucțiune **catch** și evitând astfel duplicarea codului. De exemplu, puteți crea o mulțime de excepții care reprezintă erori asociate unui anumit dispozitiv fizic. Dacă rutinele de tratare trebuie doar să înștiințeze utilizatorul de apariția unei erori în funcționarea dispozitivului, puteți utiliza o instrucțiune **catch** comună pentru toate excepțiile de acest tip. Rutina de tratare poate afișa pur și simplu șirul de caractere **Message**. Deoarece codul care realizează afișarea este același pentru toate excepțiile, o singură instrucțiune **catch** este suficientă pentru tratarea tuturor excepțiilor legate de dispozitiv.

Rezultatul acestui program este prezentat mai jos:

```
Exceptie de tipul ExceptA
Exceptie de tipul ExceptB
System.Exception: Exception of type System.Exception was thrown.
   at OrderMatters.Main()
```

Remarcați ordinea instrucțiunilor **catch**. Aceasta este singura ordine în care ele pot apărea. Deoarece **ExceptB** derivă din **ExceptA**, instrucțiunea **catch** pentru **ExceptB** trebuie să apară înaintea celei pentru **ExceptA**. Similar, rezultă că instrucțiunea **catch** pentru **Exception** (care este clasa de bază pentru toate excepțiile), trebuie să fie ultima. Pentru a vă convinge, încercați să reordonați instrucțiunile **catch**. Veți obține un mesaj de eroare la compilare.

Proiectul 10-1: Adăugarea excepțiilor la clasa Queue

În cadrul acestui proiect, veți crea două clase reprezentând excepții, care pot fi utilizate de clasele care implementează cozi din proiectul 9-1. Acestea vor reprezenta condițiile de eroare „coadă plină" și respectiv „coadă vidă". Aceste excepții sunt lansate de metodele **put()**, respectiv **get()**, atunci când se produce o eroare. Din motive de simplitate, proiectul de față va adăuga excepțiile numai pentru clasa **FixedQueue**, dar le puteți încorpora cu ușurință și în celelalte clase din proiectul 9-1.

Pas cu pas

1. Creați un fișier numit **QExcDemo.cs**.
2. În fișierul **QExcDemo.cs**, definiți următoarele excepții:


```
/*
  Proiectul 10-1
  Tratarea exceptiilor pentru clasele reprezentand cozi.
*/

using System;

// Exceptie asociata erorii "coada plina".
class QueueFullException : ApplicationException {
  public QueueFullException() : base() { }
  public QueueFullException(string str) : base(str) { }

  public override string ToString() {
    return "\n" + Message;
  }
}

// Exceptie asociata erorii "coada vida".
class QueueEmptyException : ApplicationException {
  public QueueEmptyException() : base() { }
  public QueueEmptyException(string str) : base(str) { }

  public override string ToString() {
    return "\n" + Message;
  }
}
```

Excepțiile **QueueFullException** vor fi generate atunci când se încearcă memorarea unui element într-o coadă deja plină. Excepțiile **QueueEmptyException** vor fi generate dacă se încearcă ștergerea unui element dintr-o coadă vidă.

3. Modificați clasa **FixedQueue** astfel încât să lanseze o excepție la producerea unei erori, după cum se observă mai jos. Adăugați această clasă în fișierul **QExcDemo.cs**.

```
// Coada de caractere de lungime fixata, care
// utilizeaza exceptii.
class FixedQueue : ICharQ {
  char [] q;      // tabloul care memoreaza coada
  int putloc, getloc;  // indicii put si get

  // Construieste o coada vida de dimensiune data.
  public FixedQueue(int size) {
    q = new char[size + 1];   // alocam memorie pentru coada
    putloc = getloc = 0;
  }

  // Inseram un caracter in coada.
  public void put(char ch) {
    if(putloc == q.Length - 1)
      throw new QueueFullException("Lungimea maxima este " + (q.Length - 1));

    putloc++;
    q[putloc] = ch;
  }

  // Extragem un caracter din coada.
  public char get() {
    if(getloc == putloc)
```

```
    throw new QueueEmptyException();

    getloc++;
    return q[getloc];
  }
}
```

Adăugarea excepţiilor la clasa **FixedQueue** permite tratarea erorilor într-o manieră logică. Vă amintiţi că versiunea precedentă a clasei **FixedQueue** se limita la raportarea erorii. Lansarea unei excepţii reprezintă o soluţie mult mai bună, deoarece permite codului care utilizează clasa **FixedQueue** să trateze eroarea într-o manieră adecvată.

4. Pentru a încerca noua clasă **FixedQueue**, adăugaţi clasa **QExcDemo**, prezentată mai jos, în fişierul **QExcDemo.cs**.

```
// Utilizeaza exceptiile in lucrul cu cozi.

class QExcDemo {
  public static void Main() {
    FixedQueue q = new FixedQueue(10);
    char ch;
    int i;

    try {
      // depasim intentionat capacitatea cozii
      for(i = 0; i < 11; i++) {
        Console.Write("Incercam sa inseram: " + (char) ('A' + i));
        q.put((char)('A' + i));
        Console.WriteLine(" - OK");
      }
      Console.WriteLine();
    }
    catch(QueueFullException exc) {
      Console.WriteLine(exc);
    }
    Console.WriteLine();

    try {
      // extragem intentionat mai mult de 10 elemente
      for(i = 0; i < 11; i++) {
        Console.Write("Extragem urmatorul caracter: ");
        ch = q.get();
        Console.WriteLine(ch);
      }
    }
    catch(QueueEmptyException exc) {
      Console.WriteLine(exc);
    }
  }
}
```

5. Pentru a crea programul, trebuie să compilaţi fişierul **QExcDemo.cs** împreună cu **IQChar.cs**. Acesta conţine, după cum vă amintiţi, definiţia interfeţei de coadă. La rularea programului **QExcDemo**, se va afişa următorul text:

```
Incercam sa inseram: A - OK
Incercam sa inseram: B - OK
Incercam sa inseram: C - OK
Incercam sa inseram: D - OK
Incercam sa inseram: E - OK
Incercam sa inseram: F - OK
Incercam sa inseram: G - OK
Incercam sa inseram: H - OK
Incercam sa inseram: I - OK
Incercam sa inseram: J - OK
Incercam sa inseram: K
Lungimea maxima este 10

Extragem urmatorul caracter: A
Extragem urmatorul caracter: B
Extragem urmatorul caracter: C
Extragem urmatorul caracter: D
Extragem urmatorul caracter: E
Extragem urmatorul caracter: F
Extragem urmatorul caracter: G
Extragem urmatorul caracter: H
Extragem urmatorul caracter: I
Extragem urmatorul caracter: J
Extragem urmatorul caracter:
An exception of type QueueEmptyException was thrown.
```


Utilizarea lui checked şi unchecked

Calculele aritmetice pot uneori produce depăşiri. De exemplu, să considerăm următoarea secvenţă:

```
byte a, b, result;
a = 127;
b = 127;

result = (byte)(a * b);
```

Mai sus, produsul dintre **a** şi **b** depăşeşte domeniul unei valori de tipul **byte**. Rezultatul va depăşi deci tipul lui **result**.

Limbajul C# vă permite să precizaţi dacă programul dumneavoastră va lansa sau nu o excepţie la producerea depăşirii, utilizând cuvintele cheie **checked** şi **unchecked**. Pentru a verifica dacă o expresie produce o depăşire, utilizaţi **checked**. Pentru a ignora depăşirea, utilizaţi **unchecked**. În acest caz, rezultatul va fi trunchiat astfel încât să încapă în tipul stabilit de expresie.

Instrucţiunea **checked** are două forme generale, prezentate mai jos. Prima dintre ele verifică o expresie dată. Cealaltă verifică un bloc de instrucţiuni.

checked(*expr*)

```
checked {
   // instructiuni de verificat
}
```

Mai sus, *expr* reprezintă expresia care este verificată. Dacă o expresie verificată produce o depăşire, se lansează o excepţie de tipul **OverflowException**.

Instrucțiunea **unchecked** are la rândul său două forme, prezentate mai jos. Prima formă ignoră depășirea produsă de evaluarea unei expresii date. Cea de-a doua ignoră depășirea în cadrul unui bloc de instrucțiuni.

```
unchecked(expr)

unchecked {
   // instructiuni pt. care se ignora depasirea
}
```

Mai sus, *expr* desemnează expresia pentru care nu se va verifica depășirea. Dacă se produce o depășire, rezultatul va fi trunchiat.

În continuare, iată un program care prezintă atât **checked**, cât și **unchecked**.

```
// Utilizarea lui checked si unchecked.
using System;

class CheckedDemo {
  public static void Main() {
    byte a, b;
    byte result;

    a = 127;
    b = 127;

    try {
      result = unchecked((byte)(a * b));
      Console.WriteLine("Rezultat utilizand unchecked: " + result);

      result = checked((byte)(a * b)); //se produce o exceptie
      Console.WriteLine("Rezultat utilizand checked: " + result); // nu se executa
    }
    catch(OverflowException exc) {
      // interceptam exceptia
      Console.WriteLine(exc);
    }
  }
}
```

Depășirea din această expresie produce trunchiere.

Aici, depășirea produce o excepție.

Rezultatul programului este prezentat în cele ce urmează:

```
Rezultat utilizand unchecked: 1
System.OverflowException: Exception of type
                          System.OverflowException was thrown.
                          at CheckedDemo.Main()
```

Cum este evident, expresia care nu este verificată produce o trunchiere. Expresia verificată generează o excepție.

Programul anterior a prezentat utilizarea formelor **checked** și **unchecked** în cazul unei singure expresii. Programul următor demonstrează cum puteți verifica sau ignora depășirea în cadrul unui bloc de instrucțiuni.

```
// Utilizarea lui checked si unchcked pentru blocuri de instructiuni.
using System;

class CheckedBlocks {
  public static void Main() {
    byte a, b;
    byte result;

    a = 127;
    b = 127;

    try {
      unchecked {
        a = 127;
        b = 127;
        result = unchecked((byte)(a * b));
        Console.WriteLine("Rezultat utilizand unchecked: " + result);

        a = 125;
        b = 5;
        result = unchecked((byte)(a * b));
        Console.WriteLine("Rezultat utilizand unchecked: " + result);
      }

      checked {
        a = 2;
        b = 7;
        result = checked((byte)(a * b)); // corect
        Console.WriteLine("Rezultat utilizand checked: " + result);

        a = 127;
        b = 127;
        result = checked((byte)(a * b)); // exceptie
        Console.WriteLine("Rezultat utilizand checked: " +
                          result); // nu se executa
      }
    }
    catch(OverflowException exc) {
      // interceptam exceptia
      Console.WriteLine(exc);
    }
  }
}
```


Un bloc **unchecked**.

Un bloc **checked**.

Rezultatul acestui program este prezentat mai jos:

```
Rezultat utilizand unchecked: 1
Rezultat utilizand unchecked: 113
Rezultat utilizand checked: 14
System.OverflowException: Exception of type
System.OverflowException was thrown.
at CheckedBlocks.Main()
```

După cum se observă, blocul **unchecked** determină o trunchiere, în cazul în care se produce depășirea. În cadrul blocului **checked**, se lansează o excepție.

Un motiv pentru care este necesar să utilizați **checked** și **unchecked** este că verificarea sau ignorarea depășirii este determinată de o opțiune la compilarea programului și în cadrul mediului de execuție. Pentru anumite tipuri de programe, este preferabil să precizați explicit starea de verificare a depășirii.

Sfatul expertului

Întrebare: Când trebuie să utilizăm tratarea excepţiilor într-un program? Când este cazul să definim propriile clase personalizate reprezentând excepţii?

Răspuns: Deoarece limbajul C# utilizează pe scară largă excepţii pentru raportarea erorilor, aproape toate programele reale vor apela la tratarea excepţiilor. Această parte legată de tratarea excepţiilor pare uşoară pentru majoritatea programatorilor noi în C#. Mai dificil este să decideţi când şi cum să utilizaţi propriile dumneavoastră excepţii personalizate. În general, există două modalităţi de raportare a erorilor: codurile de retur şi excepţiile. Când este fiecare dintre aceste metode mai bună decât cealaltă? Răspunsul imediat este că în C# tratarea excepţiilor este varianta optimă. Desigur, întoarcerea unui cod de eroare reprezintă o soluţie validă în unele cazuri, dar excepţiile oferă o modalitate mai puternică şi mai structurată de tratare a erorilor. Ele reprezintă modul în care programatorii profesionişti în C# tratează erorile în programele lor.

Verificarea cunoştinţelor

1. Ce clasă este rădăcina ierarhiei care reprezintă excepţii?
2. Explicaţi pe scurt modul de utilizare pentru instrucţiunile **try** şi **catch**.
3. Ce este incorect în următorul fragment?

```
// ...
vals[18] = 10;
catch(IndexOutOfRangeException exc) {
// se trateaza eroarea
}
```

4. Ce se întâmplă dacă nu interceptăm o excepţie?
5. Ce este incorect în fragmentul următor?

```
class A : Exception { ...

class B : A { ...

// ...

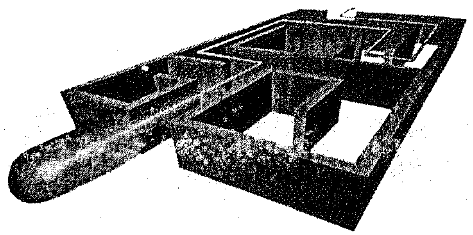
try {
// ...
}
catch(A exc) { ... }
catch(B exc) { ... }
```

6. Poate fi o excepţie interceptată de un **catch** interior relansată pentru a fi interceptată de un **catch** exterior?

7. Blocul **finally** este ultima secvenţă de cod executată înaintea terminării programului. Este adevărat sau fals? Justificaţi răspunsul.
8. În exerciţiul 3 din modulul 6, aţi creat clasa **Stack**. Adăugaţi excepţii personalizate acestei clase pentru a raporta condiţiile de „stivă plină“, respectiv de „stivă vidă“.
9. Explicaţi scopul instrucţiunilor **checked** şi **unchecked**.
10. Cum puteţi intercepta toate excepţiile?

Modulul 11

Operații de intrare-ieșire



Scopurile acestui modul

- **Prezentarea conceptului de stream**
- **Studiul claselor stream**
- **Intrări și ieșiri prin intermediul consolei**
- **Intrări și ieșiri care utilizează fișiere**
- **Citirea și scrierea datelor binare**
- **Utilizarea fișierelor cu acces aleator**
- **Conversia stringurilor în format numeric**

Încă de la începutul lucrării, ați utilizat părți ale sistemului de intrare-ieșire din C#, ca de exemplu `Console.WriteLine()`, fără însă o explicație formală detaliată. Deoarece sistemul de intrare-ieșire din C# este construit ca o ierarhie de clase, nu era posibil să prezentăm conceptele și detaliile acestui sistem fără a discuta mai întâi despre clase, moștenire și excepții. A sosit acum timpul să studiem în detaliu maniera în care sunt implementate operațiile de intrare-ieșire în C#. După cum am precizat în modulul 1, limbajul C# utilizează sistemul de intrări-ieșiri și clasele definite de arhitectura .NET. Discuția despre intrări și ieșiri în C# este deci o discuție despre sistemul de intrare-ieșire al arhitecturii .NET în general.

Acest modul studiază modul în care limbajul C# implementează intrările și ieșirile, atât prin intermediul consolei, cât și prin intermediul fișierelor. Vă prevenim de pe acum că sistemul de intrare-ieșire din C# are un volum relativ mare. Acest modul prezintă numai facilitățile cele mai importante și cel mai frecvent utilizate, dar există alte câteva aspecte legate de intrări și ieșiri pe care trebuie să le studiați în mod independent. Din fericire, sistemul de intrare-ieșire din C# este construit într-o manieră logică și strâns conectată; după ce ați înțeles aspectele fundamentale, restul sistemului este ușor de stăpânit.

Sistemul de intrare-ieșire din C# se bazează pe conceptul de stream

Programele C# efectuează operații de intrare-ieșire prin intermediul stream-urilor. Un *stream* este o entitate abstractă care fie produce, fie consumă informații. Sistemul de intrare-ieșire din C# asociază stream-urile cu dispozitivele fizice utilizate efectiv.

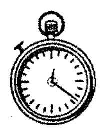
Toate stream-urile se comportă într-o manieră similară, chiar dacă dispozitivele fizice cu care sunt asociate diferă. Clasele și metodele utilizate pentru intrări și ieșiri corespund deci pentru mai multe tipuri de dispozitive. De exemplu, puteți utiliza aceleași metode pentru a scrie date atât la consolă, cât și într-un fișier de pe disc.

Stream-uri octet și stream-uri caracter

La nivelul cel mai scăzut, toate operațiile de intrare-ieșire din C# se desfășoară la nivel de octet. Aceasta este normal, deoarece majoritatea dispozitivelor sunt orientate pe octet, atunci când este vorba de operații de intrare-ieșire. În mod uzual însă, oamenii preferă să comunice utilizând caracterele. Amintiți-vă că în C# tipul `char` are 16 biți, iar `byte` are numai 8 biți. Dacă utilizați doar setul de caractere ASCII, conversia de la `char` la `byte` este foarte simplă: este suficient să ignorați octetul mai semnificativ al valorii `char`. Aceasta nu mai rămâne însă valabil și pentru restul caracterelor Unicode, care utilizează efectiv ambii octeți. Stream-urile de octeți nu reprezintă deci o alegere potrivită pentru operațiile de intrare-ieșire pe caractere. Pentru rezolvarea acestei probleme, limbajul C# definește câteva clase care convertesc un stream pe octeți într-un stream caracter, efectuând în mod automat conversiile de la `byte` la `char` și de la `char` la `byte`.

Stream-uri predefinite

Toate programele care utilizează spațiul de nume `System` dispun de trei stream-uri predefinite, accesibile prin intermediul proprietăților `Console.In`, `Console.Out` și `Console.Error`. `Console.Out` se referă la stream-ul de ieșire standard. În mod implicit, acesta este reprezentat de consola calculatorului. Atunci când apelați de exemplu `Console.WriteLine()`, informația este automat trimisă stream-ului `Console.Out`. `Console.In` reprezintă intrarea standard, care în mod implicit este tastatura. `Console.Error` se referă la stream-ul standard de eroare, care în mod implicit este tot consola. Aceste stream-uri pot fi însă redirectate către orice dispozitiv de intrare-ieșire compatibil. Stream-urile standard sunt stream-uri caracter. Prin urmare, aceste stream-uri citesc și scriu caractere.



Exerciții la minut

- Ce este un stream?
- Ce tipuri de stream-uri sunt definite în C#?
- Care sunt stream-urile predefinite?

- Un stream este o entitate abstractă, care fie produce, fie consumă informații.
- Limbajul C# definește atât stream-uri octet, cât și stream-uri caracter.
- Stream-urile predefinite sunt `Console.In`, `Console.Out` și `Console.Error`.

Clase care implementează stream-uri

Limbajul C# defineşte atât stream-uri octet, cât şi stream-uri caracter. Clasele asociate stream-urilor caracter realizează de fapt numai transformarea stream-ului octet peste care sunt implementate într-un stream caracter, efectuând automat toate conversiile. Stream-urile caracter, deşi sunt entităţi logice distincte, sunt construite pe infrastructura stream-urilor octet.

Toate clasele stream sunt definite în cadrul spaţiului de nume **System.IO**. Pentru a utiliza aceste clase, veţi include de regulă următoarea instrucţiune la începutul programului dumneavoastră:

```
using System.IO;
```

Deoarece clasa **Console** este definită în cadrul spaţiului de nume **System**, nu este nevoie să specificaţi **System.IO** şi pentru intrările şi ieşirile efectuate prin intermediul consolei.

Clasa Stream

La baza implementării conceptului de stream în C# se găseşte clasa **System.IO.Stream**. Clasa **Stream** reprezintă un stream octet şi este clasa de bază pentru toate celelalte clase stream. Această clasă este abstractă, deci nu puteţi crea instanţe de tip **Stream**. Clasa **Stream** defineşte un set de operaţii standard asupra stream-urilor. Tabelul 11-1 prezintă câteva metode frecvent utilizate definite în clasa **Stream**.

În general, dacă se produce o eroare de intrare-ieşire, metodele prezentate în tabelul 11-1 lansează o excepţie de tipul **IOException**. Dacă se încearcă o operaţie nepermisă, cum ar fi scrierea într-un stream accesibil numai la citire, se lansează o excepţie de tipul **NotSupportedException**.

Remarcaţi că în clasa **Stream** sunt definite metode atât pentru citirea, cât şi pentru scrierea datelor. Nu toate stream-urile implementează însă ambele categorii de operaţii, deoarece este posibil să creaţi stream-uri accesibile numai la citire, respectiv numai la scriere. De asemenea, nu toate stream-urile permit poziţionarea utilizând **Seek()**. Pentru a determina ce facilităţi oferă un stream, puteţi utiliza una sau mai multe proprietăţi ale clasei **Stream**. Acestea sunt prezentate în tabelul 11-2. Tot în acest tabel se găsesc şi proprietăţile **Length** şi **Position**, care conţin lungimea stream-ului şi respectiv poziţia curentă.

Tabelul 11-1 Câteva dintre metodele definite de clasa **Stream**

Metoda	Semnificaţia
`void Close()`	Închide stream-ul.
`void Flush()`	Scrie conţinutul stream-ului pe dispozitivul fizic.
`int ReadByte()`	Întoarce o reprezentare întreagă a următorului octet disponibil la citire. Întoarce –1 la întâlnirea sfârşitului de fişier.
`int Read(byte[] buf, int offset, int numBytes)`	Încearcă să citească *numBytes* octeţi în *buf*, începând de la *buf*[*offset*] şi întoarce numărul octeţilor citiţi corect.
`long Seek(long offset, SeekOrigin origin)`	Modifică poziţia curentă în stream la deplasamentul specificat de *offset*, faţă de baza specificată prin *origin*.
`void WriteByte(byte b)`	Scrie un octet într-un stream de ieşire.
`int Write(byte[] buf, int offset, int numBytes)`	Scrie un subdomeniu de *numBytes* octeţi în tabloul *buf*, începând cu *buf*[*offset*]. Întoarce numărul octeţilor scrişi.

Tabelul 11-2 Proprietăţile definite de clasa **Stream**.

Proprietatea	Semnificaţia
`bool CanRead`	Această proprietate este **true** dacă stream-ul este accesibil la citire. Proprietatea este accesibilă doar la citire.
`bool CanSeek`	Proprietatea este **true** dacă stream-ul permite cereri de poziţionare. Proprietatea este accesibilă doar la citire.
`bool CanWrite`	Această proprietate este **true** dacă stream-ul este accesibil la scriere. Proprietatea este accesibilă doar la citire.
`long Length`	Proprietatea conţine lungimea stream-ului. Această proprietate poate fi numai citită.
`long Position`	Această proprietate reprezintă poziţia curentă în cadrul stream-ului. Proprietatea este accesibilă la citire şi scriere.

Clasele care reprezintă stream-uri octet

Din clasa **Stream** derivă cele trei clase pentru reprezentarea stream-urilor octet prezentate mai jos:

Clasa Stream	Descrierea
BufferedStream	Încapsulează un stream octet, adăugând transferul prin buffer. Această tehnică îmbunătăţeşte performanţele în majoritatea cazurilor.
FileStream	Un stream octet destinat intrărilor şi ieşirilor care utilizează fişiere.
MemoryStream	Un stream octet care păstrează datele în memorie.

Puteţi de asemenea să derivaţi propriile dumneavoastră clase stream. Pentru marea majoritate a aplicaţiilor, stream-urile predefinite sunt de regulă suficiente.

Clasele care „împachetează" stream-uri caracter

Pentru crearea unui stream caracter, trebuie să „împachetaţi" un stream octet într-una din „coperţile" definite în C#. La baza ierarhiei stream-urilor caracter se află clasele abstracte **TextReader** şi **TextWriter**. Metodele definite de aceste două clase abstracte sunt disponibile în toate subclasele lor. Acestea formează deci un set minimal de funcţii de intrare-ieşire disponibile pentru toate stream-urile caracter.

Tabelul 11-3 prezintă metodele asociate operațiilor de intrare din clasa `TextReader`. În general, aceste metode lansează o excepție `IOException` în caz de eroare. (Unele dintre ele pot lansa de asemenea și alte tipuri de excepții.) Un interes sporit îl prezintă metoda `ReadLine()`, care citește o linie întreagă de text, pe care o întoarce ca `string`. Această metodă se dovedește foarte utilă atunci când aveți de citit un text care conține spații.

Tabelul 11-3 Metodele de intrare definite de clasa `TextReader`

Metoda	Semnificația
`void Close()`	Închide stream-ul de intrare.
`int Peek()`	Determină următorul caracter din stream-ul de intrare, dar nu îl șterge din stream. Întoarce –1 atunci când nu sunt disponibile caractere la intrare.
`int Read()`	Întoarce o reprezentare întreagă a următorului caracter disponibil în stream-ul de intrare apelant. Întoarce –1 dacă întâlnește sfârșitul de stream.
`int Read(char[] buf, int offset, int numChars)`	Încearcă să citească *numChars* caractere în *buf*, începând de la *buf[offset]* și întoarce numărul caracterelor citite corect.
`int ReadBlock(char[] buf, int offset, int numChars)`	Încearcă să citească *numChars* caractere în *buf*, începând de la *buf[offset]* și întoarce numărul caracterelor citite corect.
`string ReadLine()`	Citește următoarea linie de text și o întoarce sub formă de `string`. Dacă se încearcă citirea de la sfârșitul stream-ului, întoarce `null`.
`string ReadToEnd()`	Citește toate caracterele rămase în stream și le întoarce sub formă de `string`.

Clasa `TextWriter` definește versiuni ale metodelor `Write()` și `WriteLine()` care afișează toate tipurile predefinite. De exemplu, mai jos sunt numai câteva dintre versiunile supraîncărcate:

Metoda	Semnificația
`void Write(int val)`	Afișează un `int`.
`void Write(double val)`	Afișează un `double`.
`void Write(bool val)`	Afișează un `bool`.
`void WriteLine(string val)`	Afișează un `string` și trece la linie nouă.
`void WriteLine(uint val)`	Afișează un `uint` și trece la linie nouă.
`void WriteLine(char val)`	Afișează un caracter și trece la linie nouă.

Pe lângă metodele `Write()` și `WriteLine()`, `TextWriter` mai definește și metodele `Close()` și `Flush()`, prezentate mai jos:

```
virtual void Close()
virtual void Flush()
```

`Flush()` determină ca datele rămase în buffer-ul de ieșire să fie scrise pe dispozitivul fizic. `Close()` închide stream-ul.

Clasele `TextReader` și `TextWriter` sunt implementate de către clasele stream caracter prezentate mai jos. Aceste tipuri de stream-uri pun deci la dispoziție metodele și proprietățile specificate de `TextReader` și `TextWriter`.

Clasa Stream	Descrierea
`StreamReader`	Citește caractere dintr-un stream octet. Această clasă „împachetează" un stream de intrare octet.
`StreamWriter`	Scrie caractere într-un stream octet. Această clasă „împachetează" un stream de ieșire octet.
`StringReader`	Citește caractere dintr-un `string`.
`StringWriter`	Scrie caractere într-un `string`.

Stream-uri binare

Pe lângă stream-urile octet și cele caracter, limbajul C# definește două clase reprezentând stream-uri binare, care pot fi utilizate pentru a citi și respectiv pentru a scrie direct date binare. Aceste stream-uri se numesc `BinaryReader` și `BinaryWriter`. Le vom studia mai în detaliu ulterior în acest modul, când vom prezenta operațiile de intrare-ieșire din fișiere binare.

Acum, după ce ați înțeles schema generală a sistemului de intrare-ieșire din C#, restul modulului va examina în detaliu diferitele componente ale sistemului, începând cu operațiile care utilizează consola.

Exerciții la minut

- Ce clasă este la baza ierarhiei de stream-uri?
- Numiți trei proprietăți ale clasei `Stream`.
- Care sunt clasele care stau la baza stream-urilor caracter?

Intrări și ieșiri care utilizează consola

Intrările și ieșirile prin intermediul consolei sunt implementate de stream-urile standard `Console.In`, `Console.Out` și `Console.Error`. Ați utilizat operațiile de intrare și ieșire la consolă încă din modulul 1, deci în acest moment ele vă sunt desigur familiare. După cum veți observa, există și câteva facilități suplimentare față de cele studiate până acum.

Înainte de a începe, este important să subliniem un aspect prezentat anterior în cadrul lucrării: majoritatea aplicațiilor reale în C# nu au o interfață în mod text, asigurată prin intermediul consolei. De regulă, programele C# sunt aplicații grafice

- Clasa `Stream` este baza ierarhiei care reprezintă stream-uri.
- Proprietățile clasei `Stream` sunt `CanSeek`, `CanRead`, `CanWrite`, `Length` și `Position`.
- Clasele stream caracter derivă din clasele `TextReader` și `TextWriter`.

care se bazează pe un sistem de ferestre pentru asigurarea interacțiunii cu utilizatorul. Partea sistemului de intrare-ieșire din C# care corespunde operațiilor cu consola nu este deci foarte frecvent utilizată. Chiar dacă programele în mod text sunt excelente în scopuri didactice, sau la implementarea unor utilitare simple, ele nu reprezintă o soluție potrivită la dezvoltarea aplicațiilor reale.

Citirea de la consolă

Console.In este o instanță a clasei **TextReader**, deci poate accesa metodele și proprietățile definite de **TextReader**. În mod uzual, veți folosi însă metodele puse la dispoziție de clasa **Console**, care citesc automat din stream-ul **Console.In**. Clasa **Console** definește două metode pentru citirea datelor: **Read()** și **ReadLine()**.

Pentru citirea unui singur caracter, utilizați metoda **Read()**, prezentată mai jos:

```
static int Read()
```

Această metodă a fost prezentată în modulul 3. Ea întoarce următorul caracter pe care îl citește de la consolă. Valoarea întoarsă este de tipul **int** și trebuie convertită în mod explicit la **char**. În caz de eroare, metoda întoarce –1. Această metodă lansează o excepție **IOException** în caz de eșec.

Pentru citirea unui șir de caractere, utilizați metoda **ReadLine()**. Aceasta este prezentată mai jos:

```
static string ReadLine()
```

ReadLine() citește caractere până când apăsați ENTER și întoarce caracterele citite într-un obiect de tipul **string**. Și această metodă lansează o excepție **IOException** în caz de eșec.

Mai jos, prezentăm un program care citește un șir de caractere din stream-ul **Console.In**:

```
// Intrarea de la consola utilizand ReadLine().
using System;
class ReadChars {
  public static void Main() {
    string str;

    Console.WriteLine("Introduceti un sir de caractere.");
    str = Console.ReadLine(); ◄
    Console.WriteLine("Ati introdus: " + str);
  }
}
```

Citește un **string** de la tastatură.

Un exemplu de rulare a programului este prezentat mai jos:

```
Introduceti un sir de caractere.
Acesta este un test.
Ati introdus: Acesta este un test.
```

Chiar dacă metodele clasei **Console** reprezintă modalitatea cea mai simplă de citire a datelor din stream-ul **Console.In**, aveți și posibilitatea de a apela aceleași metode

utilizând explicit obiectul **TextReader**. Ca exemplu, mai jos am rescris programul precedent, astfel încât să utilizeze metodele definite de **TextReader**:

```
/* Citeste un tablou de octeti de la tastatura, utilizand
   direct Console.In. */
using System;

class ReadChars2 {
  public static void Main() {
    string str;

    Console.WriteLine("Introduceti un sir de caractere.");

    str = Console.In.ReadLine(); ◄

    Console.WriteLine("Ati introdus: " + str);
  }
}
```

Citește explicit din stream-ul **Console.In**

Remarcați modul în care metoda **ReadLine()** este invocată, utilizând direct obiectul **Console.In**. Concluzia care rezultă de aici este că, dacă doriți accesul la metodele definite de obiectul **TextReader** prin care este implementat **Console.In**, trebuie să invocați aceste metode ca în exemplul de mai sus.

Afișarea la consolă

Stream-urile **Console.Out** și **Console.Error** reprezintă obiecte de tipul **TextWriter**. Afișarea informației la consolă se realizează cel mai simplu utilizând **Write()** și **WriteLine()**, două metode cu care sunteți deja familiarizat. Există versiuni ale acestor metode pentru afișarea tuturor tipurilor predefinite. Clasa **Console** definește versiuni proprii ale metodelor **Write()** și **WriteLine()**, astfel încât acestea să poată fi apelate direct, cum s-a procedat până acum pe tot parcursul lucrării. Aveți însă opțiunea de a invoca aceste metode utilizând explicit obiectul **TextWriter** peste care sunt construite **Console.Out** și **Console.Error**.

Iată un program care prezintă afișarea la **Console.Out** și **Console.Error**:

```
// Afisarea la Console.Out si Console.Error.
using System;

class ErrOut {
  public static void Main() {
    int a = 10, b = 0;
    int result;

    Console.Out.WriteLine("Programul va genera o exceptie.");
    try {
      result = a / b; // generam exceptia
    } catch(DivideByZeroException exc) {
      Console.Error.WriteLine(exc.Message); ◄
    }
  }
}
```

Scrie în **Console.Out** și **Console.Error**.

Rezultatul programului este prezentat mai jos:

```
Programul va genera o exceptie.
Attempted to divide by zero.
```

Uneori, programatorii începători sunt derutați când se pune problema utilizării stream-ului **Console.Error**. Din moment ce atât **Console.Out**, cât și **Console.Error** afișează implicit informația pe consola sistemului, de ce sunt atunci necesare două stream-uri distincte? Răspunsul este legat de faptul că stream-urile standard pot fi redirectate către alte dispozitive. De exemplu, **Console.Error** poate fi redirectat să scrie într-un fișier de pe disc în loc să afișeze informația pe ecran. Este astfel posibil să direcționați toate erorile într-un fișier de înregistrare, fără a afecta afișarea normală la consolă. Invers, dacă redirectați numai ieșirea standard, mesajele de eroare vor putea fi vizualizate pe ecran. Vom studia redirectarea mai târziu, după ce vom prezenta operațiile de intrare și ieșire în fișiere de pe disc.

Clasa FileStream. Intrări și ieșiri orientate pe octet

Limbajul C# vă pune la dispoziție clase care vă permit să citiți și să scrieți în fișiere. Dintre acestea, cel mai frecvent întâlnite sunt fișierele memorate pe disc. La nivelul sistemului de operare, toate fișierele sunt reprezentate pe octeți. După cum vă așteptați, limbajul C# vă pune la dispoziție metode care citesc și scriu octeți din și într-un fișier. Citirea și scrierea fișierelor utilizând stream-uri octet este o operație frecvent întâlnită. Limbajul C# vă permite să „împachetați" un stream octet într-un obiect orientat pe caractere. Operațiile cu fișiere la nivel de caractere sunt utile la memorarea textului. Stream-urile caracter vor fi prezentate ulterior în acest modul. În continuare, vom descrie operațiile de intrare-ieșire la nivel de octet.

Pentru a crea un stream octet atașat unui fișier, se utilizează clasa **FileStream**. Aceasta derivă din clasa **Stream** și moștenește complet funcționalitatea clasei **Stream**.

Rețineți, clasele care reprezintă stream-uri, inclusiv **FileStream**, sunt definite în spațiul de nume **System.IO**. Trebuie deci să includeți linia:

```
using System.IO;
```

la începutul programelor care utilizează stream-uri.

Închiderea și deschiderea unui fișier

Pentru a crea un stream octet atașat unui fișier, trebuie să creați un obiect **FileStream**. Clasa **FileStream** definește mai mulți constructori. Dintre aceștia, cel mai frecvent utilizat este probabil cel prezentat mai jos:

FileStream(string *filename*, **FileMode** *mode***)**

Mai sus, *filename* reprezintă numele fișierului care va fi deschis, ce poate conține o cale complet specificată. Parametrul *mode* precizează modul în care fișierul va fi deschis. Acesta trebuie să ia una din valorile definite de enumerarea **FileMode**. Aceste valori sunt prezentate în tabelul 11-4.

Tabelul 11-4 Valorile din enumerarea FileMode

Valoarea	Semnificația
FileMode.Append	Informația se adaugă la sfârșitul fișierului.
FileMode.Create	Creează un nou fișier pentru scriere. Dacă exista deja un fișier cu același nume, acesta va fi distrus.
FileMode.CreateNew	Creează un nou fișier pentru scriere.
FileMode.Open	Deschide un fișier existent.
FileMode.OpenOrCreate	Deschide un fișier dacă acesta există, sau îl creează în caz contrar.
FileMode.Truncate	Deschide un fișier existent, dar conținutul acestuia se pierde.

Dacă se produce o eroare atunci când încercați să deschideți un fișier, se va lansa o excepție. Dacă fișierul nu poate fi deschis pentru că nu există, excepția lansată va fi de tipul **FileNotFoundException**. Dacă fișierul nu poate fi deschis din cauza unei erori de intrare-ieșire, excepția va avea tipul **IOException**. Alte excepții posibile sunt **ArgumentNullException** (dacă numele fișierului este null), **ArgumentException** (dacă parametrul *mode* este incorect), **SecurityException** (când utilizatorul nu are dreptul de acces) și **DirectoryNotFoundException** (dacă directorul specificat nu există).

Exemplul următor prezintă o modalitate de a deschide fișierul **test.dat** pentru citire:

```
FileStream fin;

try {
  fin = new FileStream("test.dat", FileMode.Open);
}
catch(FileNotFoundException exc) {
  Console.WriteLine(exc.Message);
  return;
}
catch {
  Console.WriteLine("Nu se poate deschide fisierul.");
  return;
}
```

Mai sus, prima instrucțiune **catch** interceptează eroarea „fișierul nu există". Cea de-a doua instrucțiune **catch** este „universală" și tratează toate celelalte tipuri de erori. Puteți de asemenea verifica apariția fiecărui tip de eroare în mod individual, raportând mai precis problema apărută. Din motive de simplitate, exemplele din această lucrare vor intercepta numai excepțiile **FileNotFoundException**, dar aplicațiile dumneavoastră trebuie să trateze de la caz la caz și alte tipuri de excepții.

După cum am precizat, constructorul clasei **FileStream** pe care tocmai l-am prezentat deschide un fişier atât pentru citire, cât şi pentru scriere. Dacă doriţi să limitaţi accesul numai la citire, sau numai la scriere, utilizaţi următorul constructor:

```
FileStream(string filename, FileMode mode, FileAccess how)
```

Ca şi în cazul anterior, *filename* specifică numele fişierului care va fi deschis, iar *mode* precizează modul în care se realizează deschiderea. Valoarea parametrului *how* determină modul în care fişierul va putea fi accesat. Acest parametru trebuie să aibă una din valorile definite de enumerarea **FileAccess**, valori prezentate mai jos:

```
FileAccess.Read        FileAccess.Write        FileAccess.ReadWrite
```

De exemplu, instrucţiunea următoare deschide un fişier numai pentru citire:

```
FileStream fin = new FileStream("test.dat", FileMode.Open,
FileAccess.Read);
```

Atunci când nu mai aveţi nevoie de un fişier, trebuie să îl închideţi, apelând metoda **Close()**. Forma generală a acesteia este prezentată mai jos:

```
void Close()
```

Închiderea unui fişier eliberează resursele sistemului alocate fişierului şi permite astfel utilizarea lor de către un alt fişier. Metoda **Close()** poate lansa o excepţie **IOException**.

Citirea dintr-un FileStream

Clasa **FileStream** defineşte două metode care permit citirea octeţilor dintr-un fişier: **ReadByte()** şi **Read()**. Pentru citirea unui singur octet din fişier, utilizaţi **ReadByte()**, a cărei formă generală este prezentată mai jos:

```
int ReadByte()
```

La fiecare apel, această metodă citeşte un octet din fişier şi întoarce valoarea întreagă a octetului citit. Metoda întoarce –1 dacă întâlneşte sfârşitul fişierului. În caz de eroare, metoda lansează o excepţie **IOException**.

Pentru citirea unui bloc de octeţi, utilizaţi metoda **Read()**, care are forma generală:

```
int Read(byte[] buf, int offset, int numBytes)
```

Read() încearcă să citească *numBytes* octeţi de informaţie în tabloul *buf*, începând cu poziţia *buf[offset]*. Metoda întoarce numărul octeţilor citiţi corect. Şi această metodă lansează o excepţie **IOException** în caz de eroare. Se mai pot produce şi alte câteva excepţii, printre care şi **NotSupportedException**, care este lansată dacă stream-ul nu permite operaţia de citire.

Programul următor utilizează **ReadByte()** pentru introducerea şi afişarea conţinutului unui fişier text, al cărui nume este precizat ca parametru în linia de comandă. Remarcaţi blocurile **try/catch** care tratează cele două erori ce pot apărea la prima execuţie a programului: fişierul precizat nu există, sau utilizatorul a uitat să specifice

numele fişierului. Puteţi recurge la aceeaşi metodă de fiecare dată când utilizaţi parametri în linia de comandă.

```
/* Afisarea continutului unui fisier.

   Pentru a utiliza acest program, precizati numele fisierului
   pe care doriti sa il afisati. De exemplu, pentru a vizualiza
   un fisier numit TEST.CS, utilizati urmatoarea linie de comanda:

   ShowFile TEST.CS
*/

using System;
using System.IO;

class ShowFile {
  public static void Main(string[] args) {
    int i;
    FileStream fin;

    try {
      fin = new FileStream(args[0], FileMode.Open);
    } catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message);
      return;
    } catch(IndexOutOfRangeException exc) {
      Console.WriteLine(exc.Message + "\nUtilizare: ShowFile Fisier");
      return;
    }

    // citim octeti pana intalnim EOF
    do {
      try {
        i = fin.ReadByte();
      } catch(IOException exc) {
        Console.WriteLine(exc.Message);
        return;
      }
      if(i != -1) Console.Write((char) i);
    } while(i != -1);

    fin.Close();
  }
}
```

Citim din fişier.

Dacă **i** este –1, am ajuns la sfârşitul fişierului.

Scrierea într-un fişier

Pentru scrierea unui octet într-un fişier, utilizaţi metoda **WriteByte()**. Cea mai simplă formă a acesteia este prezentată mai jos:

```
void WriteByte(byte val)
```

Această metodă scrie octetul specificat prin *val* în fişier. Dacă se produce o eroare la scriere, se lansează o excepţie **IOException**. Dacă însă stream-ul nu este deschis pentru scriere, metoda generează o excepţie de tipul **NotSupportedException**.

Puteți scrie un tablou de octeți într-un fișier apelând metoda **Write()**. Aceasta este prezentată în continuare:

```
int Write(byte[] buf, int offset, int numBytes)
```

Sfatul expertului

Întrebare: **Am observat că ReadByte() întoarce –1 atunci când întâlnește sfârșitul fișierului, dar nu întoarce nici o valoare specială în caz de eroare în fișier. De ce?**

Răspuns: În C#, erorile sunt tratate prin intermediul excepțiilor. În concluzie, dacă **ReadByte()** sau orice altă metodă care efectuează operații de intrare-ieșire întoarce o valoare, înseamnă că nu s-a produs nici o eroare. Aceasta este o modalitate mult mai sigură de tratare a erorilor de intrare-ieșire decât utilizarea codurilor speciale de eroare.

Metoda **Write()** scrie **numBytes** octeți din tabloul *buf*, începând cu poziția *buf*[*offset*], în fișier. Metoda întoarce numărul octeților pe care îi scrie. Dacă pe parcursul operației de scriere se produce o eroare, se lansează o excepție **IOException**. Dacă stream-ul nu este deschis pentru scriere, metoda generează o excepție **NotSupportedException**. Se pot produce de asemenea și alte excepții.

După cum probabil știți, atunci când se scrie într-un fișier, informația nu este imediat scrisă pe dispozitivul fizic (care este de regulă discul magnetic). Informația este memorată de sistemul de operare într-o zonă tampon, până când se adună un volum mai mare de date, care sunt scrise într-o singură operație. Această tehnică îmbunătățește performanțele sistemului de operare. De exemplu, fișierele pe disc sunt organizate în sectoare, care pot avea fiecare începând de la 128 de octeți în sus. Informația este memorată într-o zonă tampon, până când un întreg sector poate fi scris într-o singură operație. Dacă doriți să forțați scrierea datelor pe dispozitivul fizic, indiferent dacă zona tampon s-a umplut sau nu, puteți apela metoda **Flush()**, prezentată mai jos:

```
void Flush()
```

Aceasta generează o excepție **IOException** în caz de eroare.

După ce ați terminat de scris date într-un fișier, trebuie să vă amintiți să închideți fișierul, utilizând **Close()**. În acest mod, vă asigurați că informațiile rămase în zona tampon sunt scrise pe disc. Nu aveți deci nici un motiv să apelați **Flush()** înainte de a închide fișierul.

Exemplul următor copiază conținutul unui fișier în alt fișier. Numele fișierelor sursă și destinație sunt specificate în linia de comandă.


```
/* Copiaza un fisier.

   Pentru a utiliza acest program, precizati numele fisierelor sursa
   si destinatie. De exemplu, pentru a copia un fisier numit FIRST.TXT
   intr-un alt fisier numit SECOND.TXT, utilizati urmatoarea linie de comanda:

   CopyFile FIRST.TXT SECOND.TXT
*/

using System;
using System.IO;

class CopyFile {
  public static void Main(string[] args) {
    int i;
    FileStream fin;
    FileStream fout;

    try {
      // deschidem fisierul de intrare
      try {
        fin = new FileStream(args[0], FileMode.Open);
      } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message + "\nFisierul de intrare nu exista");
        return;
      }

      // deschidem fisierul de iesire
      try {
        fout = new FileStream(args[1], FileMode.Create);
      } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message + "\nEroare la deschiderea fisierului
        de iesire ");
        return;
      }
    } catch(IndexOutOfRangeException exc) {
      Console.WriteLine(exc.Message + "\nUtilizare: CopyFile Sursa Destinatie");
      return;
    }

    // Se copiaza fisierul
    try {
      do {
        i = fin.ReadByte();
        if(i != -1) fout.WriteByte((byte) i);
      } while(i != -1);
    } catch(IOException exc) {
      Console.WriteLine(exc.Message + "Eroare in fisier");
    }

    fin.Close();
    fout.Close();
  }
}
```

Citim octeți dintr-un fișier și îi scriem în altul.

Exerciţii la minut

- Ce întoarce `ReadByte()` atunci când se ajunge la sfârşitul fişierului?
- Ce rol are metoda `Flush()`?
- Ce metodă trebuie să apelaţi pentru a scrie un bloc de octeţi?

Operaţii de intrare-ieşire cu fişiere la nivel de caracter

Deşi lucrul cu fişiere la nivel de octeţi este destul de frecvent întâlnit, este posibil să utilizaţi stream-uri caracter în acest scop. Avantajul stream-urilor caracter este că ele lucrează direct cu caractere Unicode. Dacă doriţi deci să memoraţi text în format Unicode, stream-urile caracter reprezintă în mod sigur cea mai bună alegere. În general, pentru a efectua operaţii cu fişiere la nivel de caracter, trebuie să împachetaţi un obiect **FileStream** înăuntrul altui obiect de tipul **StreamReader** sau **StreamWriter**. Aceste clase realizează automat conversia unui stream octet într-un stream caracter şi invers.

Amintiţi-vă că, la nivelul sistemului de operare, fişierul constă dintr-o secvenţă de octeţi. Utilizarea obiectelor **StreamReader** şi **StreamWriter** nu modifică acest aspect.

Clasa **StreamWriter** este derivată din **TextWriter**, iar **StreamReader** este derivată din **TextReader**. Clasele **StreamWriter** şi **StreamReader** au deci acces la metodele şi proprietăţile definite în clasele lor de bază.

Utilizarea clasei StreamWriter

Pentru a crea un stream de ieşire la nivel de caractere, împachetaţi un obiect de tipul **Stream** (cum ar fi un **FileStream**) în interiorul unui obiect **StreamWriter**. Clasa **StreamWriter** defineşte mai mulţi constructori. Unul dintre cei mai cunoscuţi este prezentat mai jos:

```
StreamWriter(Stream stream)
```

Mai sus, *stream* reprezintă numele unui stream deschis. Acest constructor lansează o excepţie **ArgumentException** dacă stream-ul specificat este vid, respectiv o excepţie **ArgumentNullException** dacă *stream* este **null**. După ce este creat, un obiect **StreamWriter** realizează în mod automat conversia caracterelor la octeţi.

În continuare, prezentăm un utilitar simplu care citeşte linii de text introduse de la tastatură şi le scrie într-un fişier numit **test.txt**. Programul citeşte text până când

- `ReadByte()` întoarce –1 atunci când întâlneşte sfârşitul de fişier.
- Apelul metodei `Flush()` determină ca informaţia memorată în zona tampon să fie scrisă pe dispozitivul fizic de memorare.
- Pentru scrierea unui bloc de octeţi, se apelează metoda `Write()`.

utilizatorul introduce cuvântul „stop". În program, se utilizează un obiect **FileStream** împachetat într-un **StreamWriter** pentru scrierea informaţiei în fişier.

```
/* Un utilitar simplu de memorare a textului introdus
   de la tastatura, care utilizeaza clasa StreamWriter. */

using System;
using System.IO;

class KtoD {
  public static void Main() {
    string str;
    FileStream fout;

    try {
      fout = new FileStream("test.txt", FileMode.Create);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "Nu se poate deschide fisierul.");
      return;
    }
    StreamWriter fstr_out = new StreamWriter(fout);

    Console.WriteLine("Introduceti textul ('stop' pentru terminare).");
    do {
      Console.Write(": ");
      str = Console.ReadLine();

      if(str != "stop") {
        str = str + "\r\n";   // adaugam 'linie noua'
        try {
          fstr_out.Write(str);
        } catch(IOException exc) {
          Console.WriteLine(exc.Message + "Eroare la scrierea in fisier");
          return;
        }
      }
    } while(str != "stop");

    fstr_out.Close();
  }
}
```

Creează un obiect **StreamWriter**.

Scrie textul în fişier.

În anumite cazuri, puteţi deschide un fişier utilizând direct un obiect **StreamWriter**. Pentru aceasta, puteţi utiliza unul din următorii doi constructori:

```
StreamWriter(string filename)
StreamWriter(string filename, bool appendFlag)
```

Mai sus, *filename* precizează numele fişierului care trebuie deschis; acesta poate conţine un specificator de cale complet. Pentru cea de-a doua formă, dacă *appendFlag* este **true**, atunci textul este adăugat la sfârşitul unui fişier existent. În caz contrar, conţinutul iniţial al fişierului se pierde. În ambele situaţii, fişierul este creat dacă nu există. Ambii constructori lansează o excepţie **IOException** în cazul producerii unei erori pe parcursul operaţiei. Pot de asemenea apărea şi alte excepţii.

```
    Console.WriteLine("Sfarsitul fisierului de inregistrare.");
    log_out.Close();
  }
}
```

La rularea acestui program, pe ecran nu apare nimic. Fişierul **logfile.txt** va avea însă următorul conţinut:

```
Inceputul fisierului de inregistrare.
0
1
2
3
4
5
6
7
8
9
Sfarsitul fisierului de inregistrare.
```

Pe cont propriu, puteţi experimenta ce se întâmplă la redirectarea celorlalte stream-uri predefinite.

Proiectul 11-1: Un utilitar care compară fişiere

Acest proiect are ca scop dezvoltarea unui utilitar simplu, dar foarte folositor, de comparaţie a fişierelor. Acesta deschide două fişiere şi compară perechile corespunzătoare de octeţi. Dacă se detectează o nepotrivire, înseamnă că fişierele diferă. Dacă sfârşitul ambelor fişiere este atins simultan fără să fi apărut nepotriviri, înseamnă că fişierele sunt identice.

Pas cu pas

1. Creaţi un fişier numit **CompFiles.cs**.
2. În fişierul **CompFiles.cs**, introduceţi următorul program:

```
/*
  Proiectul 11-1

  Compararea a doua fisiere.
  Pentru a utiliza programul, specificati in linia
  de comanda numele fisierelor care trebuie comparate.

  De exemplu:
    CompFiles FIRST.TXT SECOND.TXT
*/

using System;
using System.IO;

class CompFiles {
  public static void Main(string[] args) {
    int i = 0, j = 0;
    FileStream f1;
    FileStream f2;
```



```
    try {
      // deschidem primul fisier
      try {
        f1 = new FileStream(args[0], FileMode.Open);
      } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message);
        return;
      }

      // deschidem al doilea fisier
      try {
        f2 = new FileStream(args[1], FileMode.Open);
      } catch(FileNotFoundException exc) {
        Console.WriteLine(exc.Message);
        return;
      }
    } catch(IndexOutOfRangeException exc) {
      Console.WriteLine(exc.Message + "\nUtilizare: CompFiles f1 f2");
      return;
    }

    // comparatia propriu-zisa
    try {
      do {
        i = f1.ReadByte();
        j = f2.ReadByte();
        if(i != j) break;
      } while(i != -1 && j != -1);
    } catch(IOException exc) {
      Console.WriteLine(exc.Message);
    }
    if(i != j)
      Console.WriteLine("Fisierele difera.");
    else
      Console.WriteLine("Fisierele sunt identice.");

    f1.Close();
    f2.Close();
  }
}
```

3. Pentru a încerca utilitarul **CompFiles**, copiaţi mai întâi fişierul **CompFiles.cs** într-un alt fişier, numit **temp**. Executaţi apoi linia de comandă următoare:

   ```
   CompFiles CompFiles.cs temp
   ```

   Programul trebuie să raporteze că fişierele sunt identice.

4. Comparaţi apoi fişierele **CompFiles.cs** şi **CopyFile.cs** (program prezentat anterior), utilizând această linie de comandă:

   ```
   CompFiles CompFiles.cs CopyFile.cs
   ```

   Cele două fişiere diferă, iar utilitarul **CompFiles** trebuie să raporteze aceasta.

5. În mod independent, încercaţi să adăugaţi utilitarului **CompFiles** diferite opţiuni. De exemplu, adăugaţi o opţiune care ignoră diferenţa dintre literele mari şi cele mici. O altă sugestie este ca utilitarul **CompFiles** să afişeze poziţia la care fişierele diferă.

Citirea și scrierea datelor binare

Până în momentul de față, am citit și am afișat doar octeți sau caractere, dar este posibil – și chiar destul de des – să citiți sau să scrieți și alte tipuri de date. De exemplu, puteți avea nevoie să creați un fișier care conține valori de tipuri diferite, cum ar fi `int`, `double` sau `short`. Pentru citirea și scrierea valorilor binare aparținând tipurilor predefinite în C#, trebuie să utilizați clasele `BinaryReader` și `BinaryWriter`. Atunci când utilizați aceste stream-uri, rețineți că datele se citesc și se scriu utilizând formatul intern binar, care nu poate fi citit cu ochiul liber.

BinaryWriter

Un obiect `BinaryWriter` împachetează un stream de octeți și se ocupă de scrierea informației binare. Constructorul cel mai frecvent utilizat al clasei `BinaryWriter` este cel prezentat mai jos:

`BinaryWriter(Stream` *`outputStream`*`)`

Mai sus, *outputStream* reprezintă stream-ul în care se scriu datele. Pentru a scrie date în fișier, acest parametru poate fi un obiect creat de constructorul `FileStream`. Dacă *outputStream* este `null`, se generează o excepție `ArgumentNullException`. Dacă *outputStream* nu a fost deschis pentru scriere, se lansează o excepție `ArgumentException`.

Clasa `BinaryWriter` definește metode care pot scrie toate tipurile de date predefinite în C#. Câteva dintre acestea sunt prezentate în tabelul 11-5. Clasa `BinaryWriter` definește de asemenea metodele standard `Close()` și `Flush()`, care funcționează la fel cum am prezentat anterior.

Tabelul 11-5 **Metodele de afișare cel mai frecvent utilizate definite de Clasa BinaryWriter**

Metoda	Semnificația
`void Write(sbyte val)`	Scrie un octet cu semn.
`void Write(byte val)`	Scrie un octet fără semn.
`void Write(byte[] buf)`	Scrie un tablou de octeți.
`void Write(short val)`	Scrie un întreg scurt.
`void Write(ushort val)`	Scrie un întreg scurt fără semn.
`void Write(int val)`	Scrie un întreg.
`void Write(uint val)`	Scrie un întreg fără semn.
`void Write(long val)`	Scrie un întreg lung.
`void Write(ulong val)`	Scrie un întreg lung fără semn.
`void Write(float val)`	Scrie o valoare `float`.
`void Write(double val)`	Scrie o valoare `double`.
`void Write(char val)`	Scrie un caracter.
`void Write(char[] buf)`	Scrie un tablou de caractere.
`void Write(string val)`	Scrie un `string`.

BinaryReader

Un obiect `BinaryReader` împachetează un stream de octeți și rezolvă problema citirii datelor în format binar. Constructorul cel mai des utilizat al clasei este cel prezentat mai jos:

`BinaryReader(Stream` *`inputStream`*`)`

Mai sus, *inputStream* desemnează stream-ul din care se citesc date. Pentru citirea dintr-un fișier, acest parametru poate fi o instanță creată de constructorul `FileStream`. Dacă *inputStream* este `null`, se generează o excepție `ArgumentNullException`. Dacă *inputStream* nu a fost deschis pentru citire, constructorul lansează o excepție `ArgumentException`.

Clasa `BinaryReader` pune la dispoziție metode pentru citirea tuturor tipurilor simple din C#. Cele mai frecvent utilizate sunt prezentate în tabelul 11-6. Clasa `BinaryReader` definește de asemenea și trei versiuni ale metodei `Read()`, prezentate în continuare:

`int Read()`	Întoarce o reprezentare întreagă a următorului caracter disponibil în stream-ul de intrare. Metoda întoarce –1 când întâlnește sfârșitul fișierului.
`int Read(byte[] buf, int offset, int num)`	Încearcă să citească *num* octeți în *buf*, începând cu *buf[offset]* și întoarce numărul octeților citiți corect.
`int Read(char[] buf, int offset, int num)`	Încearcă să citească *num* caractere în *buf*, începând cu *buf[offset]* și întoarce numărul caracterelor citite corect.

Aceste metode lansează o excepție `IOException` în caz de eșec. Clasa `BinaryReader` definește de asemenea și metoda standard `Close()`.

Tabelul 11-6 **Metodele de citire cel mai frecvent utilizate definite de clasa `BinaryReader`.**

Metoda	Semnificația
`bool ReadBoolean()`	Citește o valoare `bool`.
`byte ReadByte()`	Citește o valoare `byte`.
`sbyte ReadSByte()`	Citește o valoare `sbyte`.
`byte[] ReadBytes(int num)`	Citește *num* octeți și îi întoarce într-un tablou.
`char ReadChar()`	Citește o valoare `char`.
`char[] ReadChars(int num)`	Citește *num* caractere, pe care le întoarce într-un tablou.
`double ReadDouble()`	Citește o valoare `double`.
`float ReadSingle()`	Citește o valoare `float`.
`short ReadInt16()`	Citește o valoare `short`.
`int ReadInt32()`	Citește o valoare `int`.
`long ReadInt64()`	Citește o valoare `long`.
`ushort ReadUInt16()`	Citește o valoare `ushort`.
`uint ReadUInt32()`	Citește o valoare `uint`.
`ulong ReadUInt64()`	Citește o valoare `ulong`.
`string ReadString()`	Citește un `string`.

Utilizarea operațiilor de intrare-ieșire cu date în format binar

În continuare, prezentăm un program care utilizează clasele **BinaryReader** și **BinaryWriter**. Programul scrie date de diferite tipuri într-un fișier, pe care apoi le citește.

```
// Scrie si apoi citeste date in format binar.
using System;
using System.IO;
class RWData {
  public static void Main() {
    BinaryWriter dataOut;
    BinaryReader dataIn;

    int i = 10;
    double d = 1023.56;
    bool b = true;

    try {
      dataOut =
        new BinaryWriter(new FileStream("testdata", FileMode.Create));
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "\nNu se poate deschide fisierul.");
      return;
    }

    try {
      Console.WriteLine("Scriem " + i);
      dataOut.Write(i);

      Console.WriteLine("Scriem " + d);
      dataOut.Write(d);

      Console.WriteLine("Scriem " + b);
      dataOut.Write(b);

      Console.WriteLine("Scriem " + 12.2 * 7.4);
      dataOut.Write(12.2 * 7.4);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "\nEroare la scriere.");
    }

    dataOut.Close();

    Console.WriteLine();

    // acum, citim datele scrise
    try {
      dataIn = new
        BinaryReader(new FileStream("testdata", FileMode.Open));
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "\nNu se poate deschide fisierul.");
      return;
```

Scrierea datelor binare.


```
    }

    try {
      i = dataIn.ReadInt32();
      Console.WriteLine("Citim " + i);

      d = dataIn.ReadDouble();
      Console.WriteLine("Citim " + d);

      b = dataIn.ReadBoolean();
      Console.WriteLine("Citim " + b);

      d = dataIn.ReadDouble();
      Console.WriteLine("Citim " + d);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message + "\nEroare la citire.");
    }

    dataIn.Close();
  }
}
```

Citirea datelor binare scrise.

Rezultatul afișat de program este prezentat mai jos:

```
Scriem 10
Scriem 1023.56
Scriem True
Scriem 90.28
Citim 10
Citim 1023.56
Citim True
Citim 90.28
```

Exerciții la minut

- Pentru scrierea datelor în format binar, ce tip de stream trebuie să utilizați?
- Ce metodă trebuie să apelați pentru a scrie o valoare **double** în format binar?
- Ce metodă trebuie să apelați pentru a citi o valoare **short** în format binar?

Fișiere cu acces aleator

Până în prezent, am utilizat fișiere *secvențiale*, care pot fi accesate într-o manieră strict liniară, octet cu octet. Limbajul C# vă permite însă să accesați conținutul unui fișier într-o ordine aleatoare. Pentru aceasta, trebuie să utilizați metoda **Seek()**, defi-

---

- Pentru scrierea datelor în format binar, utilizați clasa **BinaryWriter**.
- Pentru a scrie o valoare **double**, apelați **Write()**.
- Pentru citirea unei valori **short**, apelați metoda **ReadInt16()**.

nită în clasa **FileStream**. Această metodă vă permite să mutați indicatorul de poziție al unui fișier (denumit și pointer) oriunde în cadrul fișierului.

Antetul metodei **Seek()** este prezentat mai jos:

```
long Seek(long newPos, SeekOrigin origin)
```

În declarația de mai sus, *newPos* specifică noua valoare a deplasamentului (exprimat în octeți) indicatorului de poziție din fișier față de locația precizată prin *origin*. Originea poate fi una din următoarele valori, definite în cadrul enumerării **SeekOrigin**:

Valoarea	Semnificația
Begin	Poziționare începând de la începutul fișierului.
Current	Poziționare începând de la poziția curentă.
End	Poziționare începând de la sfârșitul fișierului.

După apelul metodei **Seek()**, următoarele operații de citire sau de scriere în fișier vor avea loc la noua poziție. Dacă pe parcursul operației de repoziționare se produce o eroare, metoda lansează o excepție **IOException**. Dacă stream-ul utilizat nu permite operații de repoziționare, metoda generează o excepție **NotSupportedException**.

Mai jos, prezentăm un exemplu care utilizează operațiile de intrare-ieșire cu acces aleator. Programul scrie literele mari din alfabet într-un fișier, după care le citește într-o ordine aleatoare.

```
// Prezentarea accesului aleator.
using System;
using System.IO;

class RandomAccessDemo {
  public static void Main() {
  FileStream f;
    char ch;

    try {
      f = new FileStream("random.dat", FileMode.Create);
    }
    catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    // Scriem literele din alfabet.
    for(int i = 0; i < 26; i++) {
      try {
        f.WriteByte((byte)('A' + i));
      }
      catch(IOException exc) {
        Console.WriteLine(exc.Message);
        return;
      }
    }

    try {
      // citim acum anumite litere
```



```
      f.Seek(0, SeekOrigin.Begin); // pozitionare la inceput
      ch = (char) f.ReadByte();
      Console.WriteLine("Prima litera este " + ch);

      f.Seek(1, SeekOrigin.Begin); // pozitionare la al doilea octet
      ch = (char) f.ReadByte();
      Console.WriteLine("A doua litera este " + ch);

      f.Seek(4, SeekOrigin.Begin); // pozitionare la al cincilea octet
      ch = (char) f.ReadByte();
      Console.WriteLine("A cincea litera este " + ch);

      Console.WriteLine();

      // citim literele din 2 in 2.
      Console.WriteLine("Literele din 2 in 2: ");
      for(int i = 0; i < 26; i += 2) {
        f.Seek(i, SeekOrigin.Begin); // pozitionare la octetul i
        ch = (char) f.ReadByte();
        Console.Write(ch + " ");
      }
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message);
    }

    f.Close();
  }
}
```

Utilizarea lui **Seek()** pentru deplasarea pointerului.

Rezultatul afișat de acest program este afișat mai jos:

```
Prima litera este A
A doua litera este B
A cincea litera este E
Literele din 2 in 2:
A C E G I K M O Q S U W Y
```

Exerciții la minut

- Cum puteți muta indicatorul de poziție al unui fișier?
- Ce valoare a parametrului *origin* determină poziționarea față de poziția curentă din fișier?
- Ce excepție se lansează atunci când metoda **Seek()** este apelată pentru un stream care nu permite repoziționarea?

- Pentru a muta indicatorul de poziție într-un fișier, se apelează metoda **Seek()**.
- Valoarea **SeekOrigin.Current** determină poziționarea față de locația curentă din fișier.
- Dacă un stream nu permite repoziționarea, metoda **Seek()** lansează excepția **NotSupportedException**.

Conversia între stringurile numerice și reprezentarea lor internă

Înainte de a încheia capitolul dedicat operațiilor de intrare-ieșire, vom examina o tehnică foarte utilă la citirea stringurilor numerice. După cum știți deja, metoda **WriteLine()** oferă o modalitate convenabilă de afișare a diferitelor tipuri de date la consolă, inclusiv a valorilor numerice aparținând tipurilor predefinite, cum ar fi **int** sau **double**. Metoda **WriteLine()** convertește deci automat valorile numerice într-un format care poate fi citit cu ochiul liber. Limbajul nu pune însă la dispoziție o metodă care poate citi și converti stringurile care conțin valori numerice în formatul binar utilizat intern. De exemplu, nu există nici o posibilitate de a introduce de la tastatură un șir de caractere, cum ar fi „100", și de a-l converti automat în valoarea binară corespunzătoare, care poate fi memorată într-o variabilă **int**. Pentru realizarea acestei operații, trebuie să utilizați o metodă care este definită pentru toate tipurile numerice predefinite: **Parse()**.

Înainte de a începe, este necesar să precizăm un amănunt foarte important: toate tipurile predefinite din C#, cum ar fi **int** sau **double**, sunt de fapt nume alternative (aliasuri) ale unor structuri definite în cadrul arhitecturii .NET. De fapt, firma Microsoft a precizat explicit că diferențele dintre tipurile din C# și structurile definite în arhitectura .NET sunt insesizabile. Tipurile din C# reprezintă doar denumiri alternative pentru structurile arhitecturii .NET. Din cauză că tipurile valorice din C# sunt implementate cu ajutorul structurilor, fiecărui tip valoric îi corespunde un membru al unei structuri.

Pentru tipurile numerice din C#, numele structurilor corespunzătoare în arhitectura .NET și cuvintele cheie echivalente în C# sunt prezentate mai jos:

Numele structurii .NET	Numele tipului din C#
Double	double
Single	float
Int16	short
Int32	int
Int64	long
UInt16	ushort
UInt32	uint
UInt64	ulong
Byte	byte
Sbyte	sbyte

Aceste structuri sunt definite în cadrul spațiului de nume **System**. Numele complet calificat pentru **Int32** este deci **System.Int32**. Aceste structuri pun la dispoziție o gamă largă de metode, care asigură integrarea perfectă a tipurilor valorice în ierarhia de obiecte din C#. Un avantaj colateral este faptul că structurile numerice definesc metode statice care convertesc un string în format numeric în echivalentul binar corespunzător. Aceste metode de conversie sunt prezentate în continuare. Toate întorc valori binare corespunzătoare șirului de caractere pe care îl primesc ca parametru.

Structura	Metoda de conversie
Double	static double Parse(string *str*)
Single	static float Parse(string *str*)
Int64	static long Parse(string *str*)
Int32	static int Parse(string *str*)
Int16	static short Parse(string *str*)
UInt64	static ulong Parse(string *str*)
UInt32	static uint Parse(string *str*)
UInt16	static ushort Parse(string *str*)
Byte	static byte Parse(string *str*)
Sbyte	static sbyte Parse(string *str*)

Metodele **Parse()** lansează o excepție **FormatException** dacă *str* nu conține un număr corect, în conformitate cu tipul apelant. Dacă *str* este **null**, este generată excepția **ArgumentNullException**, iar dacă *str* depășește limitele tipului apelant, se lansează excepția **OverflowException**.

Metodele de conversie vă pun la dispoziție o modalitate simplă de a transforma o valoare numerică, pe care ați citit-o într-un șir de caractere de la tastatură sau dintr-un fișier text, în reprezentarea internă corespunzătoare. De exemplu, programul următor calculează media unei liste de numere introduse de utilizator. Programul cere mai întâi introducerea numărului de valori pentru care se calculează media. Programul citește numărul de valori utilizând **ReadLine()** și apelează apoi **Int32.Parse()** pentru a converti un string într-o valoare întreagă. În fine, programul citește valorile introduse, utilizând metoda **Double.Parse()** pentru a converti stringurile în valorile **double** echivalente.

```
/* Acest program calculeaza media unei liste de numere
   introduse de utilizator. */

using System;
using System.IO;

class AvgNums {
  public static void Main() {
    string str;
    int n;
    double sum = 0.0;
    double avg, t;

    Console.Write("Cate numere veti introduce: ");
    str = Console.ReadLine();
    try {
      n = Int32.Parse(str);   ← Conversia unui string la int.
    }
    catch(FormatException exc) {
      Console.WriteLine(exc.Message);
```

```
        n = 0;
      }
      catch(OverflowException exc) {
        Console.WriteLine(exc.Message);
        n = 0;
      }

      Console.WriteLine("Introduceti cele " + n + " valori.");
      for(int i = 0; i < n; i++) {
        Console.Write(": ");
        str = Console.ReadLine();
        try {
          t = Double.Parse(str);   ← Conversia unui string la double.
        } catch(FormatException exc) {
          Console.WriteLine(exc.Message);
          t = 0.0;
        }
        catch(OverflowException exc) {
          Console.WriteLine(exc.Message);
          t = 0.0;
        }
        sum += t;
      }
      avg = sum / n;
      Console.WriteLine("Media este " + avg);
    }
}
```

Iată o posibilă rulare a programului:

```
Cate numere veti introduce: 5
Introduceti cele 5 valori.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Media este 3.3
```

Utilizarea în practică a metodelor **Parse()** ne permite să îmbunătățim programul de calcul al ratelor unui împrumut, dezvoltat în cadrul proiectului 2-3. În versiunea inițială, suma împrumutată, rata dobânzii și celelalte date erau scrise direct în program. Programul devine mult mai util dacă aceste valori pot fi introduse de la tastatură. În continuare, prezentăm o versiune îmbunătățită a programului de calcul al ratei lunare, care permite introducerea datelor de la tastatură:

```
/*
  Proiectul 2-3 imbunatatit

  Calculul ratelor lunare pentru un imprumut dat.
*/

using System;

class RegPay {
  public static void Main() {
```



```
    decimal SumaTot;  // suma imprumutata
    decimal RataDob;  // rata dobanzii ca valoare zecimala, de exemplu 0.075
    decimal RatePeAn; // numarul de rate pe an
    decimal NrAni;    // numarul anilor
    decimal Rata;     // valoarea ratei lunare
    decimal numarator, numitor; // variabile temporare de lucru
    double b, e;      // baza si exponentul la apelul lui Pow()

    string str;

    Console.WriteLine("Introduceti suma imprumutata: ");
    str = Console.ReadLine();
    try {
      SumaTot = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.WriteLine("Introduceti rata dobanzii (de ex. 0.085): ");
    str = Console.ReadLine();
    try {
      RataDob = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.WriteLine("Introduceti numarul de ani: ");
    str = Console.ReadLine();
    try {
      NrAni = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    Console.WriteLine("Introduceti numarul de rate pe an: ");
    str = Console.ReadLine();
    try {
      RatePeAn = Decimal.Parse(str);
    } catch(FormatException exc) {
      Console.WriteLine(exc.Message);
      return;
    }

    numarator = RataDob * SumaTot / RatePeAn;

    e = (double) -(RatePeAn * NrAni);
    b = (double) (RataDob / RatePeAn) + 1;

    numitor = 1 - (decimal) Math.Pow(b, e);

    Rata = numarator / numitor;

    Console.WriteLine("Rata este {0:C}", Rata);
  }
}
```

Iată un posibil rezultat al acestui program:

```
Introduceti suma imprumutata: 10000
Introduceti rata dobanzii (de ex. 0.085): 0.075
Introduceti numarul de ani: 5
Introduceti numarul de rate pe an: 12
Rata este $200.38
```

Sfatul expertului

Întrebare: Ce facilități mai oferă structurile care reprezintă tipuri valorice, ca de exemplu `Int32` sau `Double`?

Răspuns: Structurile care reprezintă tipuri valorice pun la dispoziție un număr de metode care permit integrarea tipurilor predefinite din C# în ierarhia de obiecte. De exemplu, toate structurile dispun de metodele `CompareTo()`, care compară valorile încapsulate; `Equals()`, care verifică egalitatea a două valori, precum și alte metode, care întorc valoarea obiectului în diferite formate. Structurile numerice conțin de asemenea și câmpurile `MinValue` și `MaxValue`, care memorează valorile minime și maxime care pot fi memorate de un obiect având tipul corespunzător.

Proiectul 11-2: Crearea unui sistem de help memorat pe disc

În cadrul proiectului 4-1, ați creat clasa `Help`, care afișează informații despre instrucțiunile de control din C#. În acea primă versiune, informația de help era memorată direct în cadrul clasei, iar utilizatorul selecta o opțiune dintr-un meniu numerotat. Chiar dacă această soluție funcționează corect, ea nu este optimă. Pentru a adăuga sau modifica informația de help, trebuie să modificați direct codul sursă al programului. Selecția unui element utilizând un număr de ordine și nu un nume este greoaie și în același timp nepotrivită pentru listele lungi de elemente. În continuare, vom elimina acest inconvenient, creând un sistem de help memorat pe disc.

Sistemul de help memorat pe disc își păstrează informația într-un fișier de help. Acesta este un fișier text obișnuit, care poate fi modificat sau extins ori de câte ori este necesar, fără a modifica programul. Utilizatorul poate obține informațiile referitoare la un anumit subiect introducând numele subiectului dorit. Dacă acesta există, programul afișează informația corespunzătoare.

Pas cu pas

1. Trebuie mai întâi să creați fișierul de help care va fi utilizat de program. Fișierul de help este un fișier text obișnuit, organizat astfel:

```
#subiect1
informatii
#subiect2
informatii
.
.
.
```



```
#subiectN
informatii
```

Numele fiecărui subiect trebuie să fie precedat de simbolul # și trebuie să ocupe o linie separată. Această convenție permite programului regăsirea rapidă a începutului pentru orice subiect. După numele subiectului, poate urma un număr arbitrar de linii care detaliază acel subiect. Între sfârșitul unui subiect și începutul următorului trebuie însă să existe o linie vidă. De asemenea, liniile de text nu trebuie să conțină spații în partea finală.

Mai jos, prezentăm un fișier simplu de help care poate fi utilizat pentru a testa sistemul cu memorare pe disc. Fișierul conține informații despre instrucțiunile de control din C#.

```
#if
if(conditie) instructiune;
else instructiune;

#switch
switch(expresie) {
  case constanta:
    secventa de instructiuni
    break;
    // ...
  }

#for
for(initializare; conditie; iteratie) instructiune;

#while
while(conditie) instructiune;

#do
do {
  instructiune;
} while(conditie);

#break
break;

#continue
continue;

#goto
goto eticheta;
```

Denumiți acest fișier `helpfile.txt`.

2. Creați un fișier numit `FileHelp.cs`.
3. Creați o nouă clasă `Help`, care începe cu următoarele linii de cod:

```
class Help {
  string helpfile;   // numele fisierului de help
  public Help(string fname) {
    helpfile = fname;
  }
}
```

Numele fișierului de help este transmis constructorului clasei `Help` și apoi memorat în variabila instanță `helpfile`. Deoarece fiecare instanță a clasei `Help` deține o copie proprie a variabilei `helpfile`, fiecare instanță poate utiliza un alt fișier. În concluzie, pentru domenii diferite puteți crea seturi diferite de fișiere de help.

4. Adăugați metoda `helpon()`, prezentată mai jos, la clasa `Help`. Această metodă caută informația de documentare pentru un anumit subiect.

```
// afiseaza informatia corespunzatoare unui subiect dat.
public bool helpon(string what) {
  StreamReader helpRdr;
  int ch;
  string topic, info;

  try {
    helpRdr = new StreamReader(helpfile);
  }
  catch(FileNotFoundException exc) {
    Console.WriteLine(exc.Message);
    return false;
}

  try {
    do {
      // citim caractere pana gasim un #
      ch = helpRdr.Read();

      // verificam daca am gasit subiectul
      if(ch == '#') {
        topic = helpRdr.ReadLine();
        if(what == topic) { // am gasit subiectul
          do {
            info = helpRdr.ReadLine();
            if(info != null) Console.WriteLine(info);
          } while((info != null) && (info != ""));
          helpRdr.Close();
          return true;
        }
      }
    } while(ch != -1);
  }
  catch(IOException exc) {
    Console.WriteLine(exc.Message);
  }
  helpRdr.Close();
  return false;  // nu am gasit subiectul
}
```

Fișierul de help este deschis utilizând un obiect `StreamReader`. Deoarece acesta conține text, utilizarea unui stream caracter permite ca sistemul de help să fie gestionat mult mai eficient.

Metoda `helpon()` funcționează astfel: parametrul `what` primește ca valoare un `string`, care conține numele subiectului căutat. Metoda deschide fișierul de help. Metoda parcurge apoi secvențial fișierul, căutând o coincidență între parametrul `what` și un subiect din fișier. Rețineți că în fișier, fiecare subiect este precedat de semnul #, deci bucla de căutare parcurge fișierul, urmărind caracterele #. Atunci când se întâlnește un #, metoda testează coincidența între subiectul care urmează după # și cel memorat de `what`. În caz de coincidență, metoda afișează informația asociată subiectului și întoarce valoarea `true`. Dacă nu găsește subiectul până la sfârșitul fișierului, metoda întoarce `false`.

5. Clasa `Help` pune la dispoziție o metodă numită `getSelection()`, prezentată mai jos:

```
// Cauta un subiect in help.
public string getSelection() {
  string topic = "";

  Console.Write("Subiectul: ");
  try {
    topic = Console.ReadLine();
  }
  catch(IOException exc) {
    Console.WriteLine(exc.Message);
    return "";
  }
  return topic;
}
```

Această metodă cere utilizatorului să introducă numele unui subiect, citește subiectul și îl întoarce modulului apelant.

6. Listingul complet al programului care implementează un sistem de help memorat pe disc este prezentat mai jos:

```
/*
  Proiectul 11-2

  Un program care utilizeaza un fisier pe disc
  pentru memorarea informatiei de help.
*/

using System;
using System.IO;

/* Clasa Help deschide fisierul de help, cauta un subiect si apoi afiseaza
   informatia asociata subiectului respectiv. */
class Help {
  string helpfile;   // numele fisierului de help

  public Help(string fname) {
    helpfile = fname;
  }

  // afiseaza informatia corespunzatoare unui subiect dat.
  public bool helpon(string what) {
    StreamReader helpRdr;
    int ch;
    string topic, info;

    try {
      helpRdr = new StreamReader(helpfile);
    }
    catch(FileNotFoundException exc) {
      Console.WriteLine(exc.Message);
      return false;
    }
```

```
    try {
      do {
        // citim caractere pana gasim un #
        ch = helpRdr.Read();

        // verificam daca am gasit subiectul
        if(ch == '#') {
          topic = helpRdr.ReadLine();
          if(what == topic) { // am gasit subiectul
            do {
              info = helpRdr.ReadLine();
              if(info != null) Console.WriteLine(info);
            } while((info != null) && (info != ""));
            helpRdr.Close();
            return true;
          }
        }
      } while(ch != -1);
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message);
    }
    helpRdr.Close();
    return false;  // nu am gasit subiectul
  }

  // Cauta un subiect in help.
  public string getSelection() {
    string topic = "";

    Console.Write("Subiectul: ");
    try {
      topic = Console.ReadLine();
    }
    catch(IOException exc) {
      Console.WriteLine(exc.Message);
      return "";
    }
    return topic;
  }
}

// Utilizeaza sistemul de help memorat pe disc.
class FileHelp {
  public static void Main() {
    Help hlpobj = new Help("helpfile.txt");
    string topic;

    Console.WriteLine("Utilizati sistemul de help. " +
                      "Introduceti 'stop' pentru terminare.");
    do {
      topic = hlpobj.getSelection();
      if(! hlpobj.helpon(topic))
        Console.WriteLine("Subiectul nu exista.\n");

    } while(topic != "stop");
  }
}
```


Sfatul expertului

Întrebare: Mai devreme în cadrul acestui modul, ați menționat clasa `MemoryStream`, care utilizează memoria pentru stocarea informației. Cum putem utiliza acest tip de stream-uri?

Răspuns: Clasa **`MemoryStream`** este o implementare a clasei **`Stream`** care utilizează un tablou de octeți pentru stocarea informației de intrare sau de ieşire. Unul dintre constructorii definiți în această clasă este prezentat mai jos:

`MemoryStream(byte[]` *buf***)**

Mai sus, *buf* este un tablou de caractere care va fi utilizat ca sursă sau ca destinație a operațiilor de intrare-ieşire. Stream-urile create de acest constructor pot fi atât citite, cât şi scrise şi permit operația **`Seek()`**. Trebuie însă să rețineți să alocați tabloului *buf* spațiu suficient pentru a memora datele care vor fi direcționate în el.

Stream-urile stocate în memorie se dovedesc destul de utile în programare. De exemplu, puteți construi în avans o secvență de date dificil de afişat, pe care o memorați apoi într-un tablou până în momentul în care este necesară. Această metodă este utilă mai ales când programați într-un mediu grafic, aşa cum este Windows. Puteți de asemenea redirecta un stream standard astfel încât să citească informații dintr-un tablou. Aceasta vă poate ajuta de exemplu la încărcarea informației de test într-un program.

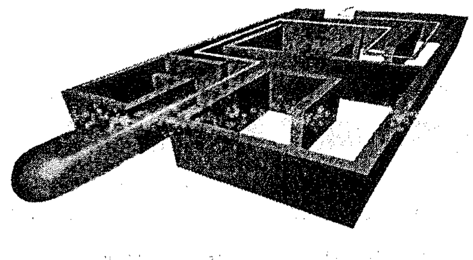
Un ultim aspect: pentru crearea unui stream caracter stocat în memorie, puteți utiliza **`StreamReader`** sau **`StreamWriter`**.

Verificarea cunoştințelor

1. De ce în C# sunt definite atât stream-uri octet, cât şi stream-uri caracter?
2. Ce clasă constituie rădăcina ierarhiei de obiecte de tip stream?
3. Arătați cum se deschide un fişier din care se pot citi octeți.
4. Arătați cum se deschide un fişier din care se pot citi caractere.
5. Ce face metoda **`Seek()`**?
6. Ce clase implementează operațiile de intrare-ieşire în mod binar pentru tipurile predefinite?
7. Ce metode se utilizează pentru redirectarea stream-urilor standard la nivelul programului?
8. Cum puteți converti un string numeric cum este „123.23" în echivalentul său binar?
9. Scrieți un program care copiază un fişier text. În urma operației de copiere, toate spațiile vor fi convertite în cratime. Utilizați în program clasele care reprezintă stream-uri octet.
10. Rescrieți programul de la ex. 9 astfel încât să utilizeze stream-uri caracter.

Modulul 12

Delegări, evenimente, spații de nume și subiecte avansate



Scopurile acestui modul

- **Însușirea conceptului de delegare**
- **Utilizarea evenimentelor**
- **Studiul spațiilor de nume**
- **Crearea operatorilor de conversie**
- **Rolul preprocesorului**
- **Studiul atributelor**
- **Pointeri și contexte nesigure**
- **Identificarea tipurilor la momentul execuției**
- **Alte cuvinte cheie ale limbajului C#**

Ajunși în acest punct, puteți constata că ați parcurs un drum lung de la începutul cărții. În acest modul final, veți studia câteva aspecte importante ale limbajului C#, printre care delegările, evenimentele și spațiile de nume. Acest modul prezintă de asemenea operatorii de conversie, atributele și rolul preprocesorului limbajului. În încheiere, pentru a asigura completitudinea lucrării, sunt prezentate câteva facilități avansate, aplicabile îndeosebi în situații speciale.

Delegări

Cei aflați la primul contact cu limbajul C# sunt uneori intimidați de conceptul de delegare, dar aceste temeri nu au un suport real. Delegările nu sunt mai dificil de înțeles sau de utilizat în comparație cu alte facilități din C#, atât timp cât v-ați format o imagine exactă despre ce reprezintă noțiunea de delegare. Într-un limbaj mai accesibil, o *delegare* este un obiect care poate referi o metodă. Așadar, atunci când creați o delegare, creați de fapt un obiect care poate memora o referință către o metodă. În plus, metoda poate fi apelată prin intermediul acestei referințe. În concluzie, o delegare poate apela metoda pe care o referă.

Chiar dacă o metodă nu este un obiect, ea ocupă o locație în memoria fizică. Aceasta este adresa punctului de intrare în metodă, fiind totodată adresa la care se transferă controlul la invocarea metodei. Adresa unei metode poate fi atribuită unei

delegări. Întrucât o delegare referă o metodă, metoda poate fi apelată prin intermediul delegării. Mai mult, aceeași delegare poate fi utilizată pentru a apela și alte metode, modificând pur și simplu metoda referită. Avantajul esențial al delegărilor este că vă permit să specificați un apel al unei metode, metoda invocată fiind însă determinată la momentul execuției și nu la compilare.

Observație

Dacă sunteți familiarizat cu limbajele C/C++, este util să știți că delegările din C# sunt similare pointerilor către funcții din C/C++.

Delegările se declară utilizând cuvântul cheie **`delegate`**. Forma generală a declarației unei delegări este prezentată mai jos:

```
delegate tip-rez nume(lista-parametri);
```

Mai sus, *tip-rez* este tipul valorii întoarse de metodele pe care delegarea le va apela. Numele delegării este specificat prin *nume*. Parametrii necesari metodelor care vor fi apelate prin intermediul delegării sunt specificați prin *lista-parametri*. După declarare, delegarea poate apela numai metodele pentru care tipul rezultatului și lista de parametri coincid cu cele specificate în declarație.

După cum am precizat, ideea de bază referitoare la delegări este că acestea se pot utiliza pentru apelul metodelor care corespund din punct de vedere al signaturii. În acest fel, determinarea metodei invocate poate fi amânată până la momentul execuției. În plus, metodele invocate pot fi atât metode instanță asociate obiectelor, cât și metode statice asociate claselor. Contează numai ca signatura metodelor să coincidă cu cea a delegării.

Pentru a vedea delegările la lucru, să începem cu exemplul simplu prezentat mai jos:

```
// Un exemplu simplu cu delegari.

using System;

// Declararea unei delegari.
delegate string strMod(string str);

class DelegateTest {
  // Inlocuieste spatiile cu liniute.
  static string replaceSpaces(string a) {
    Console.WriteLine("Inlocuim spatiile cu liniute.");
    return a.Replace(' ', '-');
  }

  // Elimina spatiile.
  static string removeSpaces(string a) {
    string temp = "";
    int i;

    Console.WriteLine("Eliminam spatiile.");
    for(i = 0; i < a.Length; i++)
```

O delegare numită **`strMod`**.

```
    if(a[i] != ' ') temp += a[i];

  return temp;
}

// Inverseaza un string.
static string reverse(string a) {
  string temp = "";
  int i, j;

  Console.WriteLine("Inversam sirul.");
  for(j = 0, i = a.Length - 1; i >= 0; i--, j++)
    temp += a[i];
  return temp;
}

public static void Main() {
  // Construim delegarile.
  strMod strOp = new strMod(replaceSpaces);   ← Construirea unei delegări.
  string str;

  // apelul metodelor utilizand delegari.
  str = strOp("Acesta este un test.");   ← Apelul unei metode prin intermediul delegării.
  Console.WriteLine("Sirul rezultat: " + str);
  Console.WriteLine();

  strOp = new strMod(removeSpaces);
  str = strOp("Acesta este un test.");
  Console.WriteLine("Sirul rezultat: " + str);
  Console.WriteLine();

  strOp = new strMod(reverse);
  str = strOp("Acesta este un test.");
  Console.WriteLine("Sirul rezultat: " + str);
 }
}
```

Rezultatul acestui program este prezentat mai jos:

```
Inlocuim spatiile cu liniute.
Sirul rezultat: Acesta-este-un-test.

Eliminam spatiile:
Sirul rezultat: Acestaesteuntest.

Inversam sirul.
Sirul rezultat: .tset nu etse atsecA
```

Să studiem acest program în amănunt. Programul declară o delegare denumită **strMod** care primeşte un parametru **string** şi întoarce tot un **string.** În clasa **DelegateTest**, sunt declarate trei metode statice, ale căror signaturi coincid cu signatura delegării. Aceste metode efectuează anumite modificări asupra stringurilor primite ca parametru. Remarcaţi că metoda **replaceSpaces()** utilizează una dintre metodele clasei **string**, numită **Replace()**, pentru înlocuirea spaţiilor cu liniuţe.

În **Main()**, se creează o referinţă de tipul **strMod** numită **strOp**, căreia i se atribuie o referinţă către metoda **replaceSpaces()**. Priviţi cu atenţie linia următoare:

```
strMod strOp = new strMod(replaceSpaces);
```

Remarcaţi modul în care metoda **replaceSpaces()** este transmisă ca parametru. Se utilizează numai numele metodei, lista de parametri nefiind specificată. Prin generalizare, se deduce că atunci când instanţiaţi o delegare, trebuie să precizaţi numai numele metodei pe care o va referi delegarea. Signatura metodei specificate trebuie de asemenea să coincidă cu declaraţia delegării. În caz contrar, se produce o eroare la compilarea programului.

În continuare, metoda **replaceSpaces()** este apelată prin intermediul instanţei delegare **strOp**, după cum se observă mai jos:

```
str = strOp("Acesta este un test.");
```

Deoarece **strOp** referă metoda **replaceSpaces()**, aceasta va fi metoda invocată.

Mai apoi, lui **strOp** i se atribuie o referinţă către metoda **removeSpaces()**, după care delegarea **strOp** este din nou apelată . De data aceasta, metoda invocată este **removeSpaces()**.

În final, lui **strOp** i se atribuie o referinţă către **reverse()**, după care delegarea este apelată. Aceasta determină un apel al metodei **reverse()**.

Ideea esenţială evidenţiată de acest exemplu este că invocarea delegării **strOp** determină apelul metodei pe care delegarea o referă la momentul invocării. Determinarea metodei apelate se rezolvă deci la momentul execuţiei şi nu la compilare.

Chiar dacă în exemplul precedent s-au utilizat metode statice, delegările pot de asemenea referi şi metode instanţă. Aceasta trebuie însă să se facă prin intermediul unei referinţe către un obiect. De exemplu, mai jos am rescris exemplul precedent, încapsulând operaţiile asupra stringurilor într-o clasă, denumită **StringOps**:

```
// Delegarile pot referi si metode instanta.

using System;

// Declararea unei delegari.
delegate string strMod(string str);

class StringOps {
  // Inlocuieste spatiile cu liniute.
  public string replaceSpaces(string a) {
    Console.WriteLine("Inlocuim spatiile cu liniute.");
    return a.Replace(' ', '-');
  }

  // Elimina spatiile.
  public string removeSpaces(string a) {
    string temp = "";
    int i;

    Console.WriteLine("Eliminam spatiile.");
```

```
    for(i = 0; i < a.Length; i++)
      if(a[i] != ' ') temp += a[i];

    return temp;
  }

  // Inverseaza un string.
  public string reverse(string a) {
    string temp = "";
    int i, j;

    Console.WriteLine("Inversam sirul.");
    for(j = 0, i = a.Length - 1; i >= 0; i--, j++)
      temp += a[i];

    return temp;
  }
}

class DelegateTest {
  public static void Main() {
    StringOps so = new StringOps();

    // Construim delegarile.
    strMod strOp = new strMod(so.replaceSpaces);
    string str;

    // apelul metodelor utilizand delegari.
    str = strOp("Acesta este un test.");
    Console.WriteLine("Sirul rezultat: " + str);
    Console.WriteLine();

    strOp = new strMod(so.removeSpaces);
    str = strOp("Acesta este un test.");
    Console.WriteLine("Sirul rezultat: " + str);
    Console.WriteLine();

    strOp = new strMod(so.reverse);
    str = strOp("Acesta este un test.");
    Console.WriteLine("Sirul rezultat: " + str);
  }
}
```

Construim o delegare utilizând o metodă instanţă.

Acest program produce acelaşi rezultat ca şi primul, dar în acest caz delegarea referă metodele utilizând o instanţă a clasei **StrOps**.

Multicasting

Una din cele mai interesante facilităţi oferite de delegări este capacitatea de *multicasting*. În termeni simpli, noţiunea de multicasting desemnează capacitatea de a crea un lanţ de metode care vor fi automat apelate la invocarea unei delegări. Un astfel de lanţ este foarte simplu de creat. Este suficient să instanţiaţi o delegare, utilizând apoi operatorul += pentru a adăuga metode în lanţ. Pentru a elimina o metodă din lanţ, utilizaţi operatorul -=. (Se pot utiliza de asemenea operatorii +, - şi = în mod independent pentru a aduna şi a scădea delegări, dar += şi -= sunt mult mai comod de folosit.) Singura restricţie este că delegările multicast trebuie să întoarcă un rezultat de tipul **void**.

Mai jos, prezentăm un exemplu de multicasting. Acesta rescrie exemplele precedente, modificând tipul întors de metodele de lucru cu stringuri la **void** şi utilizând un parametru **ref** pentru a întoarce stringul modificat în modulul apelant.

```
// Prezinta notiunea de multicasting.

using System;

// Declararea unei delegari.
delegate void strMod(ref string str);

class StringOps {
  // Inlocuieste spatiile cu liniute.
  static void replaceSpaces(ref string a) {
    Console.WriteLine("Inlocuim spatiile cu liniute.");
    a = a.Replace(' ', '-');
  }

  // Elimina spatiile.
  static void removeSpaces(ref string a) {
    string temp = "";
    int i;

    Console.WriteLine("Eliminam spatiile.");
    for(i = 0; i < a.Length; i++)
      if(a[i] != ' ') temp += a[i];

    a = temp;
  }

  // Inverseaza un string.
  static void reverse(ref string a) {
    string temp = "";
    int i, j;

    Console.WriteLine("Inversam sirul.");
    for(j = 0, i = a.Length - 1; i >= 0; i--, j++)
      temp += a[i];

    a = temp;
  }

  public static void Main() {
    // Construim delegarile.
    strMod strOp;
    strMod replaceSp = new strMod(replaceSpaces);
    strMod removeSp = new strMod(removeSpaces);
    strMod reverseStr = new strMod(reverse);
    string str = "Acesta este un test.";

    // cream o delegare multicast
    strOp = replaceSp;
    strOp += reverseStr;
```

Crearea unei delegări multicast.

```
    // Apelul delegarii multicast
    strOp(ref str);
    Console.WriteLine("Sirul rezultat: " + str);
    Console.WriteLine();

    // Eliminam inlocuirea cu liniute si adaugam eliminarea spatiilor.
    strOp -= replaceSp;
    strOp += removeSp;
    str = "Acesta este un test."; // reinitializam sirul

    // Apelul delegarii multicast
    strOp(ref str);
    Console.WriteLine("Sirul rezultat: " + str);
    Console.WriteLine();
  }
}
```

Crearea unui multicast diferit.

Programul afişează rezultatul de mai jos:

```
Inlocuim spatiile cu liniute.
Inversam sirul.
Sirul rezultat: .tset-nu-etse-atsecA

Inversam sirul.
Eliminam spatiile.
Sirul rezultat: .tsetnuetseatsecA
```

În metoda **Main()**, se creează patru instanţe de tip delegare. Prima dintre ele, **strOp**, este iniţial vidă. Celelalte trei referă fiecare câte o anumită metodă care modifică şirul de caractere. Se creează apoi o delegare multicast care apelează metodele **removeSpaces()** şi **reverse()**. Aceasta se realizează în cadrul următoarelor linii:

```
strOp = replaceSp;
strOp += reverseStr;
```

Mai întâi, lui **strOp** i se atribuie o referinţă către **replaceSp**. Utilizând apoi operatorul +=, se adaugă şi delegarea **reverseStr**. Atunci când se invocă **strOp**, se vor apela ambele metode, înlocuind spaţiile cu liniuţe şi inversând apoi şirul, după cum demonstrează şi textul afişat.

În continuare, se elimină delegarea **replaceSp** din lanţ, utilizând linia:

```
strOp -= replaceSp;
```

după care se adaugă **removeSp**, utilizând linia:

```
strOp += removeSp;
```

Se invocă apoi din nou delegarea **strOp**. De data aceasta, se inversează şirul şi se elimină spaţiile.

De ce sunt necesare delegările?

Deşi exemplele anterioare prezintă modul în care puteţi lucra cu delegări, ele nu justifică pe deplin necesitatea acestora. În general, delegările sunt utile din două motive principale. În primul rând, după cum veţi observa în secţiunea următoare,

delegările permit implementarea evenimentelor. În al doilea rând, delegările oferă programelor dumneavoastră o modalitate de a executa o metodă fără a şti care este acea metodă încă de la momentul compilării. Această capacitate se dovedeşte foarte utilă atunci când doriţi să creaţi o arhitectură care să permită adăugarea componentelor pe parcurs. De exemplu, gândiţi-vă la un program de desenare (similar într-o oarecare măsură cu utilitarul standard Paint din Windows). Utilizând o delegare, puteţi permite utilizatorului să adauge filtre speciale de culoare sau analizoare de imagine. Mai mult, utilizatorul poate crea secvenţe de astfel de filtre sau analizoare. O astfel de organizare poate fi implementată cu uşurinţă utilizând delegări.

Exerciţii la minut

- Ce este o delegare? Care este avantajul principal al utilizării delegărilor?
- Cum se poate declara o delegare?
- Ce reprezintă noţiunea de multicasting?

Evenimente

Pe fundamentul reprezentat de delegări, limbajul C# a construit o altă facilitate importantă: *evenimentele*. Un eveniment reprezintă, în esenţă, înştiinţarea automată atunci când o anumită acţiune a avut loc. Evenimentele funcţionează după următorul mecanism: un obiect care este interesat de un eveniment îşi înregistrează o rutină de tratare a acelui eveniment. Atunci când evenimentul se produce, se apelează toate rutinele înregistrate. Rutinele de tratare a evenimentelor sunt reprezentate prin delegări.

Evenimentele sunt entităţi membre ale unei clase şi se declară utilizând cuvântul cheie **event**. Forma generală a declaraţiei este prezentată mai jos:

```
event delegare-event nume-obiect;
```

Mai sus, *delegare-event* reprezintă numele delegării utilizate pentru tratarea evenimentului, iar *nume-obiect* este numele instanţei eveniment create. Vom începe prezentând un exemplu foarte simplu:

```
// Un caz simplu de utilizare a evenimentelor.

using System;

  // Declaram o delegare pentru un eveniment.
```

- O delegare este un obiect care referă o metodă. O delegare vă permite să apelaţi o metodă care este cunoscută abia la momentul execuţiei.
- Delegările se declară utilizând cuvântul cheie **delegate**, urmat de tipul rezultatului, numele delegării şi lista de parametri.
- Conceptul de multicasting permite apelul unui lanţ de metode invocând o singură delegare.

```
delegate void MyEventHandler(); ◄── Crearea unei delegări pentru eveniment.

// Declaram o clasa eveniment.
class MyEvent {
  public event MyEventHandler activate; ◄── Declararea evenimentului.

  // metoda apelata la lansarea evenimentului.
  public void fire() {
    if(activate != null)
      activate(); ◄── Lansarea evenimentului.
  }
}

class EventDemo {
  static void handler() {
    Console.WriteLine("Evenimentul s-a produs.");
  }

  public static void Main() {
    MyEvent evt = new MyEvent(); ◄── Crearea instanței eveniment.

    // se adauga handler() la lista de evenimente
    evt.activate += new MyEventHandler(handler); ◄── Adăugarea rutinei de tratare în lanț.

    // se lanseaza evenimentul
    evt.fire(); ◄── Generarea unui eveniment.
  }
}
```

Programul va afişa următorul rezultat:

```
Evenimentul s-a produs.
```

Deşi este foarte simplu, acest program conţine toate elementele esenţiale pentru activitatea de tratare a evenimentelor. Să îl examinăm cu atenţie.

Programul începe cu declaraţia unei delegări pentru tratarea evenimentului, declaraţie prezentată mai jos:

```
delegate void MyEventHandler();
```

Toate evenimentele sunt activate prin intermediul unei delegări. Delegarea asociată evenimentului defineşte deci signatura evenimentului. În acest caz, nu există parametri, dar evenimentele pot accepta parametri. Deoarece evenimentele au de obicei delegări multicast, tipul rezultatului întors trebuie să fie **void**.

În continuare, se creează o clasă eveniment, denumită **MyEvent**. În interiorul acestei clase, se defineşte o instanţă eveniment numită **activate**, utilizând linia următoare:

```
public event MyEventHandler activate;
```

Remarcaţi sintaxa folosită. Acesta este modul în care se declară toate tipurile de evenimente.

Tot în cadrul clasei **MyEvent** apare şi metoda **fire()**, care va fi apelată de un program pentru a semnala un eveniment. Metoda apelează rutina de tratare a evenimentului prin intermediul delegării **activate**, ca mai jos:


```
if(activate != null)
  activate();
```

Remarcaţi că rutina este apelată dacă şi numai dacă **activate** nu este **null**. Deoarece alte module ale programului trebuie să îşi declare interesul faţă de un eveniment pentru a fi înştiinţate la apariţia acestuia, este posibil ca metoda **fire()** să fie apelată înainte de înregistrarea rutinei de tratare. Pentru a împiedica apelul unui obiect **null**, delegarea evenimentului trebuie verificată pentru a vă asigura că nu este **null**.

În clasa **EventDemo**, se creează o rutină de tratare numită **handler()**. În acest exemplu simplu, rutina de tratare afişează pur şi simplu un mesaj; alte rutine vor executa acţiuni mult mai semnificative. În **Main()**, se creează o instanţă **MyEvent**, iar metoda **handler()** este înregistrată ca rutină de tratare a evenimentului, ca mai jos:

```
MyEvent evt = new MyEvent();
// se adauga handler() la lista de evenimente
evt.activate += new MyEventHandler(handler);
```

Remarcaţi că rutina de tratare este adăugată utilizând operatorul +=. Evenimentele permit doar operatorii += şi -=.

În fine, evenimentul este lansat după cum se observă mai jos:

```
// se lanseaza evenimentul
evt.fire();
```

Apelul metodei **fire()** determină apelul tuturor rutinelor de tratare înregistrate. În acest caz, există o singură rutină înregistrată, dar pot exista mai multe, după cum se va vedea în secţiunea următoare.

Un exemplu de eveniment multicast

Evenimentele pot fi multicast. Aceasta permite ca mai multe obiecte să poată răspunde la înştiinţarea apariţiei unui eveniment. Mai jos, este prezentat un exemplu de eveniment multicast:

```
// Un exemplu de eveniment multicast.

using System;

// Declaram o delegare pentru un eveniment.
delegate void MyEventHandler();

// Declaram o clasa eveniment.
class MyEvent {
  public event MyEventHandler activate;

  // metoda apelata la lansarea evenimentului.
  public void fire() {
    if(activate != null)
      activate();
  }
}
```

```
class X {
  public void Xhandler() {
    Console.WriteLine("Eveniment primit de un obiect X");
  }
}

class Y {
  public void Yhandler() {
    Console.WriteLine("Eveniment primit de un obiect Y");
  }
}

class EventDemo {
  static void handler() {
    Console.WriteLine("Eveniment primit de EventDemo");
  }

  public static void Main() {
    MyEvent evt = new MyEvent();
    X xOb = new X();
    Y yOb = new Y();

    // se adauga rutinele de tratare la lista de evenimente
    evt.activate += new MyEventHandler(handler);
    evt.activate += new MyEventHandler(xOb.Xhandler);
    evt.activate += new MyEventHandler(yOb.Yhandler);

    // se lanseaza evenimentul
    evt.fire();
    Console.WriteLine();

    // eliminam o rutina de tratare
    evt.activate -= new MyEventHandler(xOb.Xhandler);
    evt.fire();
  }
}
```

Crearea unui lanţ multicast de evenimente.

Textul afişat de program este prezentat mai jos:

```
Eveniment primit de EventDemo
Eveniment primit de un obiect X
Eveniment primit de un obiect Y

Eveniment primit de EventDemo
Eveniment primit de un obiect Y
```

Acest exemplu creează două clase suplimentare, numite **X** şi **Y**, care definesc la rândul lor rutine de tratare compatibile cu delegarea **MyEventHandler**. Aceste rutine pot deci deveni ele însele membre ale lanţului asociat evenimentului. Remarcaţi că rutinele de tratare din clasele **X** şi **Y** nu sunt statice. Aceasta înseamnă că trebuie create instanţe ale ambelor clase, după care rutinele asociate acestor instanţe se adaugă în lanţul asociat evenimentului.

Este important să înţelegeţi că evenimentele sunt destinate instanţelor particulare şi nu claselor în general. Fiecare instanţă a unei clase trebuie deci să se înregistreze pentru a fi înştiinţată de producerea unui eveniment. De exemplu, programul următor distribuie un eveniment către trei instanţe de tipul **X**:


```
// Evenimentele sunt destinate instantelor, nu claselor.

using System;

// Declaram o delegare pentru un eveniment.
delegate void MyEventHandler();

// Declaram o clasa eveniment.
class MyEvent {
  public event MyEventHandler activate;

  // metoda apelata la lansarea evenimentului.
  public void fire() {
    if(activate != null)
      activate();
  }
}

class X {
  int id;

  public X(int x) { id = x; }

  public void Xhandler() {
    Console.WriteLine("Eveniment primit de instanta " + id);
  }
}

class EventDemo {
  public static void Main() {
    MyEvent evt = new MyEvent();
    X o1 = new X(1);
    X o2 = new X(2);
    X o3 = new X(3);

    evt.activate += new MyEventHandler(o1.Xhandler);
    evt.activate += new MyEventHandler(o2.Xhandler);
    evt.activate += new MyEventHandler(o3.Xhandler);

    // se lanseaza evenimentul
    evt.fire();
  }
}
```

Rezultatul afişat de acest program este prezentat mai jos:

```
Eveniment primit de instanta 1
Eveniment primit de instanta 2
Eveniment primit de instanta 3
```

După cum se arată în acest rezultat, fiecare obiect îşi declară interesul pentru eveniment în mod separat şi fiecare primeşte o înştiinţare separată.

```
  }
}

class NSDemo {
  public static void Main() {
    // CountDown poate fi acum utilizata direct
    CountDown cd1 = new CountDown(10);
    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    CountDown cd2 = new CountDown(20);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Clasa **CountDown** se poate utiliza acum direct.

Programul demonstrează şi un alt aspect important: utilizarea unui nou spaţiu de nume nu ascunde celelalte spaţii de nume vizibile în mod curent. Atunci când un nou spaţiu de nume devine vizibil, numele definite în cadrul său se adaugă la cele definite în celelalte spaţii vizibile în mod curent. În concluzie, în programul anterior, ambele spaţii **System** şi **Counter** sunt vizibile.

A doua formă a lui `using`

Directiva **using** are şi o a doua formă, care este prezentată mai jos:

using *alias* = *nume*;

În acest caz, *alias* devine un alt nume pentru clasa sau spaţiul de nume specificat prin *nume*. În continuare, am rescris încă o dată programul de numărare, astfel încât să creăm un alias pentru **Counter.CountDown** numit **Count**.

```
// Forma lui using cu alias.
using System;

// cream un alias pentru Counter.CountDown.
using Count = Counter.CountDown;

// Se declara un spatiu de nume pentru operatia de numarare.
```

Creăm un alias pentru **Counter.CountDown**.


```
namespace Counter {
  // O clasa care numara invers.
  class CountDown {
    int val;

    public CountDown(int n) { val = n; }

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

class NSDemo {
  public static void Main() {
    Count cd1 = new Count(10);
    int i;

    do {
      i = cd1.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    Count cd2 = new Count(20);

    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    cd2.reset(4);
    do {
      i = cd2.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();
  }
}
```

Utilizăm aliasul.

După ce **Count** a fost declarat ca alias pentru **Counter.CountDown**, poate fi utilizat la declararea obiectelor fără a fi necesară nici o altă calificare cu spaţiul de nume. De exemplu, în program, linia:

```
Count cd1 = new Count(10);
```

creează un obiect **CountDown**.

Spațiile de nume sunt aditive

Pot exista mai multe declarații **namespace** cu același nume. Aceasta permite distribuirea unui spațiu de nume în mai multe fișiere, sau chiar separarea sa în cadrul aceluiași fișier. Spre exemplu, programul următor definește două spații de nume numite **Counter**. Primul conține clasa **CountDown**. Celălalt conține clasa **CountUp**. La compilare, conținutul ambelor spații de nume **Counter** este adăugat laolaltă.

```
// Spatiile de nume sunt aditive.
using System;

// Se "descopera" spatiul Counter.
using Counter;

// Primul spatiu Counter.
namespace Counter {
  // O clasa care numara invers.
  class CountDown {
    int val;

    public CountDown(int n) { val = n; }

    public void reset(int n) {
      val = n;
    }

    public int count() {
      if(val > 0) return val--;
      else return 0;
    }
  }
}

// Un alt spatiu Counter.
namespace Counter {
  // Un numarator crescator.
  class CountUp {
    int val;
    int target;

    public int Target { get { return target; } }

    public CountUp(int n) { target = n; val = 0; }

    public void reset(int n) {
      target = n;
      val = 0;
    }

    public int count() {
      if(val < target) return val++;
      else return target;
    }
  }
}
```

Spațiul **Counter** este declarat de două ori.


```
class NSDemo {
  public static void Main() {
    CountDown cd = new CountDown(10);
    CountUp cu = new CountUp(8);
    int i;

    do {
      i = cd.count();
      Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    do {
      i = cu.count();
      Console.Write(i + " ");
    } while(i < cu.Target);

  }
}
```

Acest program afișează următorul rezultat:

```
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8
```

Remarcați și un alt aspect: instrucțiunea

```
using Counter;
```

dezvăluie întreg conținutul spațiului de nume **Counter**. Ambele clase, **CountDown** și **CountUp**, pot fi deci referite direct, fără o calificare cu spațiul de nume. Faptul că spațiul de nume **Counter** a fost divizat în două nu are nici o importanță.

Spațiile de nume pot fi imbricate

Este posibil ca un spațiu de nume să fie imbricat în interiorul altuia. Să considerăm programul:

```
// Spatiile de nume pot fi imbricate.
using System;

namespace NS1 {
  class ClassA {
    public ClassA() {
      Console.WriteLine("Constructorul clasei ClassA");
    }
  }

  namespace NS2 { // un spatiu de nume imbricat
    class ClassB {
      public ClassB() {
        Console.WriteLine("Constructorul clasei ClassB");
      }
    }
  }
}
```

NS2 este inclus în **NS1**.

```
class NestedNSDemo {
  public static void Main() {
    NS1.ClassA a = new NS1.ClassA();

// NS2.ClassB b = new NS2.ClassB(); // Eroare! NS2 nu e vizibil direct

    NS1.NS2.ClassB b = new NS1.NS2.ClassB(); // corect
  }
}
```

Programul afişează următorul rezultat:

```
Constructorul clasei ClassA
Constructorul clasei ClassB
```

În program, spaţiul de nume **NS2** este cuprins în interiorul lui **NS1**. Ca urmare, pentru a referi clasa **ClassB**, aceasta trebuie calificată atât cu **NS1**, cât şi cu **NS2**. Nu este suficient să o calificăm doar cu **NS2**. După cum se observă, numele spaţiilor sunt separate prin caracterul punct (.).

Este posibil să specificaţi un spaţiu imbricat printr-o singură instrucţiune **namespace**, separând numele fiecărui spaţiu printr-un punct. De exemplu:

```
namespace NSExterior {
namespace NSInterior {
// -
}
}
```

poate fi specificată şi astfel:

```
namespace NSExterior.NSInterior {
// -
}
```

Spaţiul implicit de nume

Dacă nu declaraţi nici un spaţiu de nume în programul dumneavoastră, se va utiliza spaţiul implicit de nume. Din acest motiv, programele din modulele anterioare nu au avut nevoie de declaraţii **namespace**. Deşi spaţiul implicit este comod de utilizat pentru exemplele scurte din această carte, majoritatea aplicaţiilor reale vor fi conţinute într-un spaţiu propriu de nume. Motivul principal care stă la baza încapsulării codului dumneavoastră într-un spaţiu de nume este evitarea conflictelor de nume. Spaţiile de nume constituie un alt instrument de care dispuneţi pentru organizarea programelor şi creşterea viabilităţii acestora în mediile distribuite moderne.

Exerciţii la minut

- Un spaţiu de nume defineşte un ________
- Care sunt cele două forme ale lui **using**?
- Este posibil ca un spaţiu de nume să fie împă

Proiectul 12-1: Includerea tipul într-un spaţiu de nume propriu

În cadrul proiectului 7-1, aţi creat o clasă nur un tip mulţime. Acesta este exact tipul de clasă l includeţi într-un spaţiu de nume propriu. Motivu **Set** poate intra cu uşurinţă în conflict cu alte clase care au acel un spaţiu de nume propriu, puteţi evita eventualele conflicte d

Pas cu pas

1. Utilizând codul din fişierul **SetDemo.cs** (fişier creat în cadrul întreaga clasă **Set** într-un fişier separat, numit **Set.cs**. Inclu interiorul spaţiului de nume **MyTypes.Set**, după cum se obs

```
/*
  Proiectul 12-1

  Includerea clasei Set in spatiul de nume propriu.
*/
using System;

namespace MyTypes.Set {

  class Set {
    char [] members;  // tabloul care memoreaza multim
    int len;          // numarul de elemente

    // Construieste o multime vida.
    public Set() {
      len = 0;
    }

    // Construieste o multime fara elemente cu o dime
    public Set(int size) {
      members = new char[size]; // aloca memorie pent
      len = 0;                  // multimea nu are de
    }

    // Construieste o multime "clonand" o alta multim
    public Set(Set s) {
      members = new char[s.len];  // aloca memorie pe
      for(int i = 0; i < s.len; i++) members[i] = s[i
```

- Un spaţiu de nume defineşte un domeniu de valabilitate.
- Cele două forme ale directivei **using** sunt **using nume;** şi u
- Da.

```
    len = s.len;    // numarul de elemente
  }

  // Implementam proprietatea Length, accesibila numai la citire.
  public int Length {
    get {
      return len;
    }
  }

  // Implementam indexarea, accesibila numai la citire.
  public char this[int idx] {
    get {
      if(idx >= 0 & idx < len) return members[idx];
      else return (char)0;
    }
  }

  /* Stabileste daca un element face parte dintr-o multime.
     Intoarce indexul elementului, sau -1 daca acesta nu este gasit. */
  int find(char ch) {
    int i;

    for(i = 0; i < len; i++)
      if(members[i] == ch) return i;

    return -1;
  }

  // Adauga un element la o multime.
  public static Set operator+(Set ob, char ch) {
    Set newset = new Set(ob.len + 1);

    // copiem elementele
    for(int i = 0; i < ob.len; i++)
      newset.members[i] = ob.members[i];

    // stabilim valoarea lui len
    newset.len = ob.len;

    // verificam daca elementul exista deja in multime
      if(ob.find(ch) == -1)  {// daca nu am gasit elementul, adaugam
      // adauga elementul la noua multime
      newset.members[newset.len] = ch;
      newset.len++;
    }
    return newset;    // intoarce multimea actualizata
  }

  // Elimina un element din multime.
  public static Set operator-(Set ob, char ch) {
    Set newset = new Set();
    int i = ob.find(ch); // i = -1 daca elementul nu exista

    // copiem si compactam elementele ramase dupa eliminare
    for(int j = 0; j < ob.len; j++)
      if(j != i) newset = newset + ob.members[i];
```



```
      return newset;
    }

    // Reuniunea multimilor.
    public static Set operator+(Set ob1,
      Set newset = new Set(ob1);    // "cl

      // adaugam pe rand elementele din c
      for(int i = 0; i < ob2.len; i++)
        newset = newset + ob2[i];

      return newset;// intoarce multimea
    }

    // Diferenta multimilor.
    public static Set operator-(Set ob1,
      Set newset = new Set(ob1);    // "c.

      // eliminam pe rand elementele cel
      for(int i = 0; i < ob2.len; i++)
        newset = newset - ob2[i];

      return newset;// intoarce multimea
    }
  }
}
```

2. După ce ați inclus clasa **Set** în spațiul de num **MyTypes.Set** să fie inclus în toate programele

```
using MyTypes.Set;
```

3. O altă posibilitate este calificarea completă de mai jos:

```
MyTypes.Set.Set s1 = new MyTypes.Set.Set
```

Operatori de conversi

În anumite situații, este necesar să utili: care implică alte tipuri de date. În unele c: cărcarea unuia sau mai multor operatori. Î simpla conversie a unui tip clasă la un alt limbajul C# vă permite să creați un *operat* transformă obiectele aparținând unei clase

Există două forme de operatori de con expliciți. Forma generală pentru fiecare ca

```
public static operator implicit tip-d
public static operator explicit tip-d
```

Mai sus, *tip-destinatie* este tipul în care a se efectuează conversia, iar *value* este valo

de conversie întorc numai valori de tipul *tip-destinatie*, nefiind permis nici un alt specificator de tip.

Dacă operatorul de conversie este specificat ca fiind **implicit**, atunci conversia este invocată automat, de exemplu atunci când se utilizează un obiect într-o expresie împreună cu tipul destinație. Dacă operatorul de conversie conține cuvântul cheie **explicit**, atunci conversia este invocată numai când se folosește un cast. Date fiind tipul sursă și tipul destinație, nu puteți defini simultan operatori de conversie implicită și explicită.

Pentru a prezenta utilizarea operatorilor de conversie, vom reveni la clasa **TreiD** pe care am creat-o în modulul 7. Amintiți-vă că obiectele **TreiD** memorează coordonate tridimensionale. Să presupunem că doriți să convertiți un obiect de tipul **TreiD** într-o valoare întreagă, care să poată fi utilizată într-o expresie întreagă. Conversia va avea loc prin înlocuirea instanței **TreiD** cu produsul celor trei dimensiuni. Pentru aceasta, veți scrie un operator implicit de conversie care arată astfel:

```
public static implicit operator int(TreiD op1)
{
  return op1.x * op1.y * op1.z;
}
```

Mai jos este prezentat un program care utilizează acest operator de conversie:

```
// Un exemplu care utilizeaza un operator de conversie implicita.
using System;

// O clasa care reprezinta coordonate tridimensionale.
class TreiD {
  int x, y, z;  // coordonate 3D

  public TreiD() { x = y = z = 0; }
  public TreiD(int i, int j, int k) {x = i; y = j; z = k; }

  // Supraincarcam operatorul binar +.
  public static TreiD operator +(TreiD op1, TreiD op2)
  {
    TreiD result = new TreiD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // O conversie implicita de la TreiD la int.
  public static implicit operator int(TreiD op1)    <-- Un operator de conversie de la TreiD la int.
  {
    return op1.x * op1.y * op1.z;
  }

  // Afiseaza coordonatele X, Y, Z.
  public void show()
  {
```



```
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TreiDDemo {
  public static void Main() {
    TreiD a = new TreiD(1, 2, 3);
    TreiD b = new TreiD(10, 10, 10);
    TreiD c = new TreiD();
    int i;

    Console.Write("Punctul a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Punctul b: ");
    b.show();
    Console.WriteLine();

    c = a + b; // adunam a si b
    Console.Write("Rezultatul a + b: ");
    c.show();
    Console.WriteLine();

    i = a;  // conversie la int    <-- Invocarea conversiei la int.
    Console.WriteLine("Rezultatul conversiei i = a: " + i);
    Console.WriteLine();

    i = a * 2 - b;  // conversie la int    <-- Invocarea conversiei la int.
    Console.WriteLine("Rezultatul expresiei a * 2 - b: " + i);
  }
}
```

Programul afișează următorul rezultat:

```
Punctul a: 1, 2, 3

Punctul b: 10, 10, 10

Rezultatul a + b: 11, 12, 13

Rezultatul conversiei i = a: 6

Rezultatul expresiei a * 2 - b: -988
```

După cum arată acest program, atunci când un obiect **TreiD** este utilizat într-o expresie întreagă, ca de exemplu **i = a**, se aplică operatorul de conversie acelui obiect. În acest caz particular, conversia întoarce valoarea 6, care este produsul coordonatelor memorate în **a**. Dacă o expresie nu necesită conversia la **int**, operatorul de conversie nu este apelat. Din acest motiv, expresia **c = a + b** nu invocă metoda **operator int()**.

Rețineți că puteți crea operatori de conversie diferiți pentru a rezolva diferite probleme. Puteți de exemplu să definiți un alt operator care realizează conversia la **double** sau la **long**. Fiecare dintre acești operatori se aplică automat și independent de ceilalți.

Operatorii impliciți de conversie sunt apelați automat atunci când conversia este necesară într-o expresie, la transmiterea unui obiect ca parametru pentru o metodă, în instrucțiunile de atribuire și de asemenea atunci când se utilizează un cast explicit către tipul destinație. O altă posibilitate este de a crea un operator explicit de conversie, care va fi invocat numai la utilizarea unui cast explicit. Operatorii de conversie explicită nu sunt invocați automat. De exemplu, mai jos am rescris programul anterior, utilizând un operator explicit de conversie la **int**:

```
// Utilizarea unei conversii explicite.
using System;

// O clasa care reprezinta coordonate tridimensionale.
class TreiD {
  int x, y, z;  // coordonate 3D

  public TreiD() { x = y = z = 0; }
  public TreiD(int i, int j, int k) {x = i; y = j; z = k; }

  // Supraincarcam operatorul binar +.
  public static TreiD operator +(TreiD op1, TreiD op2)
  {
    TreiD result = new TreiD();

    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;

    return result;
  }

  // Conversia este acum explicita.
  public static explicit operator int(TreiD op1)   <- Conversia este acum explicită.
  {
    return op1.x * op1.y * op1.z;
  }

  // Afiseaza coordonatele X, Y, Z.
  public void show()
  {
    Console.WriteLine(x + ", " + y + ", " + z);
  }
}

class TreiDDemo {
  public static void Main() {
    TreiD a = new TreiD(1, 2, 3);
    TreiD b = new TreiD(10, 10, 10);
    TreiD c = new TreiD();
    int i;

    Console.Write("Punctul a: ");
    a.show();
    Console.WriteLine();
    Console.Write("Punctul b: ");
    b.show();
```



```
    Console.WriteLine();

    c = a + b; // adunam a si b
    Console.Write("Rezultatul a + b: ");
    c.show();
    Console.WriteLine();

    i = (int)a;  // conversie explicita la int -- castul este obligatoriu   <- Casturile sunt acum obligatorii.
    Console.WriteLine("Rezultatul conversiei i = a: " + i);
    Console.WriteLine();

    i = (int)a * 2 - (int)b;  // casturile sunt obligatorii   <- Casturile sunt acum obligatorii.
    Console.WriteLine("Rezultatul expresiei a * 2 - b: " + i);
  }
}
```

Deoarece operatorul de conversie este acum marcat ca fiind explicit, conversia la **int** trebuie forțată prin intermediul unui cast. De exemplu, pe linia de mai jos:

```
i = (int)a;  // conversie explicita la int -- castul este obligatoriu
```

dacă eliminați castul, programul nu se va mai putea compila.

Sfatul expertului

Întrebare: Din moment ce conversiile implicite sunt invocate automat, fără necesitatea unui cast, care este rolul conversiilor explicite?

Răspuns: Deși se efectuează mult mai comod, conversiile implicite trebuie utilizate numai în situațiile în care, în mod firesc, operațiile de conversie nu determină erori. Pentru aceasta, conversiile implicite trebuie create numai atunci când se îndeplinesc două condiții. Prima condiție este ca să nu aibă loc nici o pierdere de informație prin trunchiere, depășire sau pierderea semnului. A doua condiție este ca operația de conversie să nu genereze o excepție. Dacă o operație de conversie nu poate satisface aceste două condiții, trebuie să fie explicită.

Operatorii de conversie presupun câteva restricții:

- Nu puteți defini conversii pentru tipurile predefinite. De exemplu, nu puteți redefini conversia de la **double** la **int**.
- Nu puteți defini conversii având tipul sursă sau tipul destinație **object**.
- Date fiind tipul sursă și tipul destinație, nu puteți defini simultan operatori de conversie implicită și de conversie explicită.
- Nu puteți defini o conversie a unei clase de bază către o clasă derivată din ea.
- Nu puteți defini conversii având tipul sursă sau tipul destinație reprezentat de o interfață.

Exerciţii la minut

- Ce este un operator de conversie?
- Ce diferenţă există între conversiile implicite şi cele explicite?
- Este posibil să definiţi un operator de conversie a unui tip predefinit?

Preprocesorul limbajului

Limbajul C# defineşte câteva *directive preprocesor*, care afectează modul în care fişierul sursă al programului este interpretat de compilator. Aceste directive afectează textul fişierului sursă în care apar, înaintea translatării programului în cod obiect. Directivele preprocesor sunt în mare măsură o reminiscenţă din C++. De fapt, preprocesorul limbajului C# este foarte asemănător cu cel definit în C++. Termenul de *directivă preprocesor* provine de la faptul că aceste instrucţiuni erau iniţial interpretate într-o fază separată a compilării, numită *preprocesare*. Tehnologiile moderne de compilare utilizate în prezent nu mai necesită un stadiu separat de preprocesare pentru a trata aceste directive, dar numele s-a păstrat din vechime.

Limbajul C# defineşte următoarele directive preprocesor:

```
#define       #elif      #else      #endif
#endregion    #error     #if        #line
#region       #undef     #warning
```

Toate directivele preprocesor încep cu semnul #. În plus, fiecare directivă preprocesor trebuie să se găsească pe o linie separată.

Dată fiind arhitectura modernă, orientată spre obiecte, a limbajului C#, directivele preprocesor nu mai sunt la fel de necesare ca în limbajele mai vechi. Cu toate acestea, ele pot fi valoroase din când în când, în special pentru compilarea condiţionată. În continuare, vom examina pe rând fiecare directivă.

#define

Directiva **#define** defineşte o secvenţă de caractere numită *simbol*. Existenţa sau neexistenţa unui simbol poate fi determinată de directivele **#if** sau **#elif** şi se utilizează pentru a controla compilarea. Forma generală a directivei este:

```
#define simbol
```

Remarcaţi că această instrucţiune nu se termină cu punct şi virgulă. Între **#define** şi simbol pot fi oricâte spaţii. După apariţia simbolului pe linie, aceasta se poate

---

- Operatorii de conversie precizează conversii ale tipurilor clasă.
- Conversiile implicite sunt automat invocate. Conversiile explicite trebuie invocate utilizând un cast.
- Nu.

termina numai cu caracterul linie nouă. De exemplu, pentru a defini simbolul **EXPERIMENTAL**, utilizaţi directiva:

```
#define EXPERIMENTAL
```

Sfatul expertului

Întrebare: În C/C++, se poate utiliza #define pentru efectuarea substituţiilor textuale; de exemplu, se pot asocia nume unor valori date sau se pot crea macrodefiniţii similare funcţiilor. Este permis şi în C# acest gen de utilizare a directivei #define?

Răspuns: Nu. În C#, **#define** se utilizează numai pentru a defini simboluri.

#if şi #endif

Directivele **#if** şi **#endif** vă permit compilarea condiţionată a unei secvenţe de cod, în cazul în care o expresie care conţine unul sau mai multe simboluri se evaluează la **true**. Un simbol este **true** dacă a fost definit. În caz contrar, simbolul este **false**. Cu alte cuvinte, dacă un simbol a fost definit printr-o directivă **#define**, va fi evaluat la **true**.

Forma generală a directivei **#if** este:

```
#if expresie-simbol
  secventa de instructiuni
#endif
```

Dacă expresia care urmează după **#if** este **true**, codul cuprins între directivă şi **#endif** va fi compilat. În caz contrar, secvenţa de cod este ignorată. Directiva **#endif** marchează sfârşitul unui bloc **#if**.

Cele mai simple expresii simbol conţin doar numele unui simbol. În expresiile simbol, puteţi utiliza următorii operatori: !, ==, !=, && şi ||. Utilizarea parantezelor este de asemenea permisă.

Să considerăm exemplul:

```
// Prezinta utilizarea #if, #endif si #define.
#define EXPERIMENTAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Se compileaza numai in versiunea experimentala.");
    #endif

    Console.WriteLine("Instructiune compilata in toate versiunile.");
  }
}
```

Instrucţiune compilată numai dacă simbolul **EXPERIMENTAL** este definit.

Acest program afişează următorul text:

```
Se compileaza numai in versiunea experimentala.
Instructiune compilata in toate versiunile.
```

Programul defineşte simbolul **EXPERIMENTAL**. În consecinţă, atunci când directiva **#if** este întâlnită, expresia simbol se evaluează la **true**, iar prima instrucţiune **WriteLine()** este compilată. Dacă eliminaţi definiţia simbolului **EXPERIMENTAL** şi recompilaţi programul, prima instrucţiune **WriteLine()** nu va mai fi compilată, deoarece expresia directivei **#if** se va evalua la **false**. În toate situaţiile, cea de-a doua instrucţiune **WriteLine()** este compilată, deoarece nu face parte din blocul **#if**.

După cum am precizat, o directivă **#if** poate utiliza o expresie simbol. Iată un exemplu:

```
// Utilizarea unei expresii simbol.
#define EXPERIMENTAL
#define TRIAL

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Se compileaza numai in versiunea experimentala.");
    #endif

    #if EXPERIMENTAL && TRIAL
      Console.Error.WriteLine("Testarea versiunii experimentale de proba.");
    #endif

    Console.WriteLine("Instructiune compilata in toate versiunile.");
  }
}
```

O expresie simbol.

Textul afişat de acest program este cel prezentat mai jos:

```
Se compileaza numai in versiunea experimentala.
Testarea versiunii experimentale de proba.
Instructiune compilata in toate versiunile.
```

În acest exemplu, se definesc două simboluri, **EXPERIMENTAL** şi **TRIAL**. Cea de-a doua instrucţiune **WriteLine()** este compilată numai dacă ambele simboluri sunt definite.

#else şi #elif

Directiva **#else** funcţionează aproape la fel cu clauza **else** care face parte din limbajul C#: ea oferă o alternativă, în cazul în care **#if** eşuează. Exemplul anterior poate fi extins după cum urmează:

```
// Utilizarea directivei #else.

#define EXPERIMENTAL
```



```
using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Se compileaza numai in versiunea experimentala.");
    #else
      Console.WriteLine("Se compileaza in versiunea finala.");
    #endif

    #if EXPERIMENTAL && TRIAL
      Console.Error.WriteLine("Testarea versiunii experimentale de proba.");
    #else
      Console.Error.WriteLine("Nu este o versiune experimentala de proba.");
    #endif

    Console.WriteLine("Instructiune compilata in toate versiunile.");
  }
}
```

Utilizarea directivei **#else**.

Rezultatul afişat este prezentat mai jos:

```
Se compileaza numai in versiunea experimentala.
Nu este o versiune experimentala de proba.
Instructiune compilata in toate versiunile.
```

Deoarece simbolul **TRIAL** nu mai este definit, se va utiliza porţiunea **#else** a celei de-a doua secvenţe condiţionale de cod.

Remarcaţi că **#else** marchează atât sfârşitul blocului **#if**, cât şi începutul blocului **#else**. Aceasta este necesar, deoarece unei directive **#if** i se poate asocia un singur **#endif**.

Directiva **#elif** are semnificaţia de „altfel dacă" şi se utilizează la construirea unei scări **if-else-if** pentru cazul opţiunilor multiple la compilare. Directiva **#elif** este urmată de o expresie simbol. Dacă expresia este adevărată, blocul de cod care urmează este compilat fără a se mai testa şi alte expresii **#elif**. În caz contrar, se trece la verificarea următorului bloc din secvenţă. Forma generală a directivei **#elif** este:

```
#if expresie-simbol
  secventa de instructiuni
#elif expresie-simbol
  secventa de instructiuni
#elif expresie-simbol
  secventa de instructiuni
#elif expresie-simbol
  secventa de instructiuni
#elif expresie-simbol
.
.
.
#endif
```

De exemplu:

```
// Prezentarea directivei #elif.
#define RELEASE

using System;

class Test {
  public static void Main() {

    #if EXPERIMENTAL
      Console.WriteLine("Se compileaza numai in versiunea experimentala.");
    #elif RELEASE
      Console.WriteLine("Se compileaza numai in versiunea finala.");
    #else
      Console.WriteLine("Se compileaza pentru testarea interna.");
    #endif

    #if TRIAL && !RELEASE
      Console.WriteLine("Versiune de test.");
    #endif

    Console.WriteLine("Se compileaza in toate versiunile.");
  }
}
```

Folosirea lui **#elif**.

Textul afişat este cel prezentat mai jos:

```
Se compileaza numai in versiunea finala.
Se compileaza in toate versiunile.
```

#undef

Directiva **#undef** elimină o definiţie anterioară a numelui macro care îi urmează. Cu alte cuvinte, macrodefiniţia devine „nedefinită". Forma generală a directivei **#undef** este:

```
#undef simbol
```

De exemplu:

```
#define SMALL

#if SMALL
// ...
#undef SMALL
// in acest punct, SMALL este nedefinit.
```

După apariţia directivei **#undef**, simbolul **SMALL** nu mai este definit.

Directiva **#undef** se utilizează în principal pentru a permite localizarea simbolurilor la acele secţiuni de cod care au nevoie de ele.

#error

Directiva **#error** forţează compilatorul să oprească procesul de compilare. Această directivă se utilizează pentru depanarea programelor. Forma generală a directivei **#error** este:

```
#error mesaj-eroare
```

Atunci când directiva **#error** este întâlnită, se afişează mesajul de eroare. De exemplu, atunci când compilatorul întâlneşte linia:

```
#error Eroare la verificare!
```

compilarea este oprită şi se afişează mesajul de eroare „Eroare la verificare".

#warning

Directiva **#warning** este similară cu **#error**, dar generează un avertisment şi nu o eroare. Compilarea nu este deci oprită. Forma generală a directivei **#warning** este:

```
#warning mesaj-avertisment
```

#line

Directiva **#line** stabileşte numărul liniei şi numele fişierului pentru fişierul care o conţine. Numărul şi numele vor fi apoi utilizate pentru toate erorile şi avertismentele generate pe parcursul compilării. Forma generală a directivei **#line** este:

```
#line numar „nume-fisier"
```

unde *numar* poate fi orice întreg pozitiv şi devine noul număr de linie, iar parametrul opţional *nume-fisier* poate fi orice identificator valid de fişier, care va deveni noul nume al fişierului. Directiva **#line** se utilizează în primul rând pentru depanare, cât şi în unele aplicaţii speciale.

#region şi #endregion

Directivele **#region** şi **#endregion** vă permit să definiţi o regiune care poate fi extinsă sau restrânsă în mediul integrat Visual C++, atunci când utilizaţi facilitatea de prezentare sub formă de schemă. Forma generală a acestor directive este prezentată mai jos:

```
#region
  // secventa de cod
#endregion
```

Exerciții la minut

- Cum puteți defini un simbol?
- Care este rolul directivei **#if**?
- Ce operatori se pot utiliza în expresiile **#if** sau **#elif**?

Atribute

Limbajul C# vă permite să adăugați informații declarative în programe sub formă de *atribute*. Un atribut definește o informație adițională asociată unei clase, structuri, metode etc. Spre exemplu, puteți defini un atribut care determină tipul butoanelor afișate de o anumită clasă.

Atributele sunt specificate între paranteze pătrate, înaintea elementului căruia îi sunt asociate. Puteți defini propriile atribute, sau puteți utiliza atributele definite de limbajul C#. Crearea unor atribute proprii depășește cadrul acestei lucrări, dar este foarte simplu să utilizați două atribute predefinite în C#: **Conditional** și **Obsolete**. Acestea sunt studiate în cele ce urmează.

Atributul `Conditional`

Atributul **Conditional** este probabil cel mai interesant atribut predefinit din C#. Acest atribut vă permite să definiți metode condiționale. Metodele condiționale sunt invocate numai dacă un anumit simbol a fost definit printr-o directivă **#define**. În caz contrar, metoda este ignorată. Metodele condiționale reprezintă deci o alternativă la compilarea condiționată utilizând directiva **#if**. Pentru a utiliza atributul **Conditional**, trebuie să includeți spațiul de nume **System.Diagnostics**.

Vom începe prin prezentarea unui exemplu:

```
// Prezinta utilizarea atributului Conditional.
#define TRIAL

using System;
using System.Diagnostics;

class Test {

  [Conditional("TRIAL")]
  void trial() {
    Console.WriteLine("Versiune de test, nu este distribuita.");
  }
```

trial() se execută numai dacă a fost definit **TRIAL**.

- Pentru a defini un simbol, se utilizează **#define**.
- Directiva **#if** precede un bloc de cod care se compilează numai dacă expresia simbol asociată cu **#if** este adevărată. Blocul este terminat cu **#endif**.
- Operatorii permiși în expresiile **#if** și **#elif** sunt **!=**, **==**, **&&**, **||** și **!**.


```
  [Conditional("RELEASE")]
  void release() {
    Console.WriteLine("Versiunea finala.");
  }

  public static void Main() {
    Test t = new Test();

    t.trial();   // apelul are loc numai daca simbolul TRIAL este definit
    t.release(); // apelata numai daca este definit simbolul RELEASE
  }
}
```

release() se execută numai dacă a fost definit **RELEASE**.

Textul afișat de program este prezentat mai jos:

```
Versiune de test, nu este distribuita.
```

Să studiem mai în amănunt acest program, pentru a înțelege de ce afișează acest text. Mai întâi, remarcați că programul definește simbolul **TRIAL**. Observați apoi cum sunt scrise metodele **trial()** și **release()**. Ambele sunt precedate de atributul **Conditional**, care are forma generală:

[Conditional ***simbol*****]**

unde *simbol* este simbolul care determină dacă metoda se va executa sau nu. Acest atribut se poate utiliza numai în cazul metodelor. Dacă simbolul este definit, atunci metoda se va executa ori de câte ori va fi apelată. Dacă însă simbolul nu este definit, metoda nu va fi executată.

În corpul lui **Main()**, se apelează ambele metode **trial()** și **release()**. Singurul simbol definit este însă **TRIAL**; în consecință, se va executa numai metoda **trial()**. Apelul metodei **release()** este ignorat. Dacă definiți și simbolul **RELEASE**, metoda **release()** va fi și ea apelată. Dacă eliminați definiția lui **TRIAL**, metoda **trial()** nu va mai fi apelată.

Metodele condiționale sunt supuse câtorva restricții: în primul rând, ele trebuie să întoarcă **void**. Ele trebuie să fie membre ale unei clase și nu ale unei interfețe. Aceste metode nu pot fi precedate de cuvântul cheie **override**.

Atributul `Obsolete`

Atributul **System.Obsolete** vă permite să marcați o entitate dintr-un program atunci când nu mai este de actualitate. Atributul are forma generală:

[Obsolete ***„mesaj“*****]**

Mai sus, *mesaj* reprezintă textul afișat la compilarea respectivei entități din program. Iată un scurt exemplu:

```
// Prezentarea atributului Obsolete.
using System;

class Test {
```

```
  [Obsolete("Utilizati varianta myMeth2().")]
  static int myMeth(int a, int b) {
    return a / b;
  }

  // Versiunea imbunatatita a metodei myMeth().
  static int myMeth2(int a, int b) {
    return b == 0? 0 : a / b;
  }

  public static void Main() {
    // acest apel va afisa un avertisment
    Console.WriteLine("4 / 3 este " + Test.myMeth(4, 3));

    // nu se afiseaza nici un avertsiment
    Console.WriteLine("4 / 3 este " + Test.myMeth2(4, 3));
  }
}
```

Atunci când la compilarea programului se ajunge la apelul metodei **myMeth()** în **Main()**, se va genera un mesaj de avertisment care informează utilizatorul să folosească metoda **myMeth2()** în loc.

Exerciţii la minut

- Ce este un atribut?
- Ce este o metodă condiţională?
- Care este rolul atributului **Obsolete**?

Cod nesigur

Limbajul C# vă permite să dezvoltaţi ceea ce este cunoscut sub numele de cod „nesigur". Chiar dacă această afirmaţie poate părea şocantă, nu este. Prin cod nesigur nu înţelegem codul scris defectuos, ci codul care nu se execută sub gestiunea motorului comun de programare (MCP). După cum am precizat în modulul 1, limbajul C# se utilizează în mod obişnuit pentru elaborarea codului tratat. Este însă posibil să scrieţi cod care nu se execută sub controlul MCP. Deoarece acest cod netratat nu cade sub incidenţa controlului şi restricţiilor care caracterizează codul tratat, este denumit „nesigur", deoarece este imposibil să verificăm dacă nu cumva efectuează anumite operaţii periculoase. Termenul de *nesigur* nu înseamnă deci că acest cod este

---

- Atributele reprezintă informaţii suplimentare ataşate unor anumite entităţi, cum ar fi clasele, structurile sau metodele.
- Metodele condiţionale sunt acele metode care au atributul **Conditional** specificat. Metodele condiţionale se execută numai dacă simbolul precizat de atribut a fost definit.
- Atributul **Obsolete** determină generarea unui mesaj de avertisment, care indică utilizarea unui element care nu mai este de actualitate.

în mod inerent eronat. Acest termen spune doar că este posibil ca el să efectueze operaţii care nu cad sub incidenţa supravegherii asigurate de contextul caracteristic codului tratat.

Codul tratat, chiar dacă este avantajos în majoritatea situaţiilor, evită utilizarea *pointerilor.* Dacă aveţi experienţă de lucru în C sau C++, atunci ştiţi că pointerii sunt variabile care memorează adresele altor obiecte. Pointerii sunt deci oarecum similari cu referinţele din C#. Diferenţa principală este că un pointer poate indica orice adresă din memorie, în timp ce referinţele indică numai obiecte de acelaşi tip cu ele. Deoarece pointerii pot indica orice locaţie de memorie, utilizarea lor eronată reprezintă un fapt cât se poate de posibil. Utilizarea pointerilor poate de asemenea genera cu uşurinţă erori de programare. Din această cauză, limbajul C# nu permite utilizarea pointerilor atunci când se dezvoltă cod tratat. Pointerii sunt însă necesari şi totodată foarte utili în anumite tipuri de programe (cum ar fi utilitarele la nivelul sistemului de operare), iar limbajul C# permite crearea şi utilizarea lor. Toate operaţiile cu pointeri trebuie însă marcate ca nesigure, deoarece ele se execută în afara mediului tratat.

În general, dacă aveţi nevoie să creaţi cod care se execută în afara MCP, este mai bine să utilizaţi limbajul C++. În C#, pointerii se declară şi se utilizează la fel ca în C/C++; dacă ştiţi să utilizaţi pointeri în C/C++, atunci vă veţi descurca şi în C#. Amintiţi-vă însă că scopul limbajului C# este crearea codului tratat. Capacitatea limbajului de a lucra cu cod netratat permite utilizarea sa pentru o clasă specială de probleme, nefiind utilizată în programarea de zi cu zi în C#. Pentru compilarea codului netratat, trebuie să utilizaţi opţiunea **/unsafe** la compilare.

Lucrul cu codul netratat reprezintă un subiect avansat, discuţia acestuia în detaliu depăşind cadrul acestei lucrări. Acestea fiind zise, vom examina pe scurt lucrul cu pointeri şi cele două cuvinte cheie care permit utilizarea codului netratat: **unsafe** şi **fixed**.

O prezentare scurtă a pointerilor

Pointerii sunt variabile care memorează adresele altor obiecte. De exemplu, dacă **x** conţine adresa lui **y**, atunci **x** „îl indică" pe **y**. Variabilele pointer trebuie declarate astfel. Forma generală a declaraţiei unei variabile pointer este:

tip nume-var*;

Mai sus, *tip* reprezintă tipul de bază al pointerului; acesta trebuie să fie un tip simplu. Numele variabilei pointer este *nume-var.* De exemplu, pentru a declara variabila **ip** ca pointer la o variabilă **int**, utilizaţi declaraţia:

```
int* ip;
```

Pentru un pointer de tip **float**, scrieţi:

```
float* fp;
```

În general, într-o instrucțiune de declarare, precedarea numelui variabilei cu semnul `*` face ca variabila să devină un pointer.

Tipul datelor pe care pointerii le indică este determinat de tipul de bază al pointerilor. În exemplele precedente, **ip** poate fi utilizat pentru a indica un **int**, iar **fp** poate indica o valoare **float**.Rețineți însă că nu există nici o regulă care să împiedice un pointer să indice orice adresă din memorie. Din acest motiv, pointerii sunt potențial nesiguri.

Există doi operatori care lucrează asupra pointerilor: `*` și `&`. Operatorul unar `&` întoarce adresa în memorie a operandului său. (Amintiți-vă că toți operatorii unari necesită un singur operand.) De exemplu, secvența:

```
int* ip;
int num = 10;

ip = &num;
```

introduce în **ip** adresa de memorie a variabilei **num**. Această adresă reprezintă locația variabilei în memoria internă a calculatorului și nu are nici o legătură cu *valoarea* lui **num**. Variabila **ip** nu conține deci valoarea 10 (valoarea inițială a lui **num**), ci adresa la care **num** este memorată. Este mai ușor să rețineți că operatorul `&` „întoarce adresa" variabilei care îl precede. Instrucțiunea de atribuire anterioară poate fi deci descrisă în cuvinte prin „**ip** primește adresa lui **num**".

Cel de-al doilea operator este `*`, care este inversul lui `&`. Acest operator unar întoarce valoarea variabilei care se găsește la adresa specificată de operandul său. Continuând cu același exemplu, dacă **ip** conține adresa din memorie a variabilei **num**, atunci secvența:

```
int val;
val = *ip;
```

va atribui lui **val** valoarea 10 a lui **num**, care este indicată de **ip**. Puteți reține că operatorul `*` întoarce valoarea aflată la adresa reprezentată de operandul său. În acest caz, instrucțiunea de mai sus se poate citi „**val** primește valoarea de la adresa **ip**."

Pointerii se pot de asemenea utiliza în cazul structurilor. Atunci când accesați un membru al unei structuri prin intermediul unui pointer, trebuie să utilizați operatorul săgeată `->` în locul operatorului punct (.). De exemplu, fie structura:

```
struct MyStruct {
  public int x;
  public int y;
  public int sum() { return x + y; }
}
```

Iată cum puteți accesa membrii structurii prin intermediul unui pointer:

```
MyStruct o = new MyStruct();
MyStruct * p;   // declararea unui pointer

p = &o;
p->x = 10;
```



```
p->y = 20;

Console.WriteLine("Suma este " + p->sum());
```

Sfatul expertului

Întrebare: **Se știe că la declararea unui pointer în C++, operatorul `*` nu se distribuie unei liste de variabile din declarație. Cu alte cuvinte, instrucțiunea:**

```
int* p, q;
```

declară un pointer întreg numit p și un întreg numit q, fiind echivalentă cu următoarele două declarații:

```
int* p;
int q;
```

Aceasta rămâne valabil și în C#?

Răspuns: Nu. În C#, operatorul `*` este distributiv, ceea ce înseamnă că declarația:

```
int* p, q;
```

creează două variabile pointer. Instrucțiunea de mai sus este echivalentă cu următoarele două declarații:

```
int* p;
int* q;
```

Aceasta este o diferență de care trebuie să țineți cont atunci când portați cod din C++ în C#.

Utilizarea codului nesigur

Codul care utilizează pointeri trebuie să fie marcat ca nesigur, folosind cuvântul cheie **unsafe**. Puteți marca ca fiind nesigure atât instrucțiunile luate fiecare în parte, cât și metodele. De exemplu, programul de mai jos utilizează pointeri în metoda **Main()**, care este marcată ca fiind nesigură:

```
// Prezinta utilizarea pointerilor si codului nesigur.
using System;

class UnsafeCode {
  // marcam Main() ca fiind nesigura
  unsafe public static void Main() {
    int count = 99;
    int* p;  // cream un pointer int

    p = &count;  // punem adresa lui count in p

    Console.WriteLine("Valoarea initiala a lui count este " + *p);

    *p = 10;  // atribuim lui count o valoare utilizand p
```

Se marchează **Main()** ca **unsafe**, deoarece utilizează un pointer.

```
        Console.WriteLine("Noua valoare a lui count este " + *p);
    }
}
```

Rezultatul afişat de acest program este cel prezentat mai jos:

```
Valoarea initiala a lui count este 99
Noua valoare a lui count este 10
```

Utilizarea modificatorului `fixed`

Modificatorul **fixed** evită ca un obiect gestionat să fie deplasat în memorie de mecanismul de colectare a spaţiului disponibil. Aceasta este necesar dacă există un pointer care referă obiectul. Deoarece pointerul nu are idee de acţiunile mecanismului de colectare, dacă obiectul va fi deplasat, atunci pointerul va indica în mod eronat vechea adresă a obiectului. Forma generală a lui **fixed** este prezentată mai jos:

```
fixed (tip *p = &obFixat) {
// utilizarea obiectului fixat
}
```

Mai sus, *p* este un pointer căruia i s-a atribuit adresa unui obiect. Obiectul va rămâne în locaţia curentă de memorie până la terminarea execuţiei blocului. Ţinta lui **fixed** poate fi de asemenea o singură instrucţiune. Cuvântul cheie **fixed** poate fi utilizat numai într-un context nesigur.

Iată un exemplu care utilizează **fixed**:

```
// Prezinta utilizarea lui fixed.
using System;

class Test {
  public int num;
  public Test(int i) { num = i; }
}

class UnsafeCode {
  // marcam Main() ca fiind nesigura
  unsafe public static void Main() {
    Test o = new Test(19);

    fixed(int* p = &o.num) { // utilizam fixed pentru a memora adresa
                             // lui o.num in p

      Console.WriteLine("Valoarea initiala a lui o.num este " + *p);

      *p = 10; // atribuim o valoare lui o.num utilizand p

      Console.WriteLine("Noua valoare a lui o.num este " + *p);
    }
  }
}
```

Utilizarea lui **fixed** pentru a fixa locaţia lui **o**.

Rezultatul acestui program este prezentat mai jos:

```
Valoarea initiala a lui o.num este 19
Noua valoare a lui o.num este 10
```

Mai sus, **fixed** împiedică mutarea obiectului **o**. Întrucât **p** indică **o.num**, dacă **o** ar fi deplasat, atunci **p** ar indica o locaţie incorectă.

După cum am menţionat la începutul acestei secţiuni, crearea şi utilizarea codului nesigur constituie un subiect avansat, care implică mai multe aspecte faţă de cele discutate aici. Dacă aveţi efectiv nevoie să scrieţi cod nesigur, trebuie să studiaţi mai amănunţit acest subiect. (O tratare detaliată a acestui subiect va apărea în lucrarea „*C# | Manual complet*" a aceluiaşi autor.)

Exerciţii la minut

- Ce este un pointer?
- Când trebuie utilizat cuvântul cheie **unsafe**?
- Care este rolul lui **fixed**?

Identificarea tipurilor la execuţie

În C#, este posibil ca tipul unui obiect să fie determinat abia la momentul execuţiei. Limbajul C# conţine de fapt trei cuvinte cheie care implementează identificarea tipurilor la momentul execuţiei: **is**, **as** şi **typeof**. Chiar dacă identificarea tipurilor la momentul execuţiei este o facilitate avansată de care nu veţi beneficia în programarea de zi cu zi, este important să îi înţelegeţi mecanismul general.

Testarea unui tip utilizând `is`

Puteţi determina dacă un obiect este de un tip dat, utilizând operatorul **is**. Forma generală a acestuia este prezentată mai jos:

```
expr is tip
```

Mai sus, *expr* este o expresie al cărui tip este comparat cu *tip*. Dacă tipul lui *expr* coincide sau este compatibil cu *tip*, atunci rezultatul acestei operaţii este **true**. În caz contrar, rezultatul este **false**. Iată un exemplu:

```
// Prezinta utilizarea lui is.
using System;

class A { }
```

- Un pointer este o variabilă care conţine adresa din memorie a unui alt obiect.
- Codul care utilizează pointeri trebuie specificat ca fiind **unsafe**.
- Modificatorul **fixed** împiedică mecanismul de colectare a spaţiului disponibil să mute un obiect în memorie.

```
class B : A { }

class UseIs {
  public static void Main() {
    A a = new A();
    B b = new B();

    if(a is A) Console.WriteLine("a este de tipul A");
    if(b is A)
      Console.WriteLine("b este de tipul A, deoarece B deriva din A");
    if(a is B)
      Console.WriteLine("Nu se va afisa - tipul A nu deriva din B");

    if(b is B) Console.WriteLine("b este de tipul B");
    if(a is object) Console.WriteLine("a este de tipul Object");
  }
}
```

Acest **is** este adevărat deoarece a este de tipul A.

a nu este de tipul B, deci acest **is** este fals.

Rezultatul afişat de program este prezentat mai jos:

```
a este de tipul A
b este de tipul A, deoarece B deriva din A
b este de tipul B
a este de tipul Object
```

Majoritatea expresiilor **is** nu necesită explicaţii suplimentare, dar două dintre acestea merită o scurtă discuţie. Mai întâi, remarcaţi instrucţiunea:

```
if(b is A)
Console.WriteLine("b este de tipul A, deoarece B deriva din A");
```

Instrucţiunea **if** se execută, deoarece **b** este un obiect de tipul **B**, care este derivat din tipul **A**. Tipul lui **b** este deci compatibil cu **A**. Propoziţia inversă nu este însă adevărată. Atunci când se execută linia:

```
if(a is B)
Console.WriteLine("Nu se va afisa - tipul A nu deriva din B");
```

instrucţiunea **if** eşuează, deoarece **a** este de tipul **A**, care nu este derivat din **B**. Tipurile nu sunt deci compatibile.

Utilizarea lui as

În unele cazuri, este necesar să efectuaţi un cast la momentul execuţiei, fără însă să generaţi o excepţie în cazul în care conversia eşuează. Pentru aceasta, utilizaţi operatorul **as**, care are următoarea formă generală:

expr **as** *tip*

Mai sus, *expr* este expresia care este convertită la tipul *tip*. Dacă operaţia de conversie reuşeşte, se întoarce o referinţă de tipul *tip*. În caz contrar, operatorul întoarce o referinţă **null**.

Utilizarea lui typeof

Puteţi obţine un obiect **System.Type** corespunzător unui tip dat utilizând **typeof**, care are următoarea formă generală:

typeof(*tip***)**

Mai sus, *tip* este tipul pentru care dorim obţinerea unui obiect **System.Type**. Clasa **System.Type** prezintă informaţii asociate anumitor tipuri. Utilizând obiectul întors de **typeof**, puteţi determina anumite informaţii referitoare la tipul dat. De exemplu, programul de mai jos afişează numele complet al clasei **StreamReader**:

```
// Prezinta utilizarea lui typeof.
using System;
using System.IO;

class UseTypeof {
  public static void Main() {
    Type t = typeof(StreamReader);
    Console.WriteLine(t.FullName);
  }
}
```

Acest program afişează următorul text:

```
System.IO.StreamReader
```

Clasa **System.Type** conţine multe alte metode, câmpuri şi proprietăţi pe care le puteţi studia pe cont propriu.

Exerciţii la minut

- Care este operatorul care determină dacă un obiect este compatibil cu un tip dat?
- Ce diferenţă există între operatorul **as** şi un cast obişnuit?
- Ce întoarce operatorul **typeof**?

- Operatorul **is** determină dacă un obiect este compatibil cu un tip dat.
- Un cast lansează o excepţie atunci când eşuează. Operatorul **as** întoarce **null** în caz de eşec.
- Operatorul **typeof** obţine un obiect **System.Type** care descrie tipul specificat.

Alte cuvinte cheie

În finalul lucrării, vom prezenta pe scurt cele câteva cuvinte cheie definite în C# care nu au fost descrise în altă parte.

Modificatorul de acces internal

Pe lângă modificatorii de acces **public**, **private** și **protected**, pe care i-am utilizat pe parcursul lucrării, limbajul C# mai definește și modificatorul **internal**. Menționat pe scurt în modulul 6, **internal** este utilizat pentru a declara membrii care sunt cunoscuți în toate fișierele dintr-un grup, dar nu sunt vizibili în afara grupului. Prin *grup* înțelegem un fișier (sau o mulțime de fișiere) care conține toată informația aferentă unui program, inclusiv cea care asigură controlul versiunilor. În termeni mai simpli, putem afirma că membrii marcați ca **internal** sunt cunoscuți numai pe parcursul unui program și nicăieri în altă parte.

sizeof

În mod ocazional, este util să cunoașteți dimensiunea, în octeți, a tipurilor valorice definite în C#. Pentru a obține această informație, utilizați operatorul **sizeof**. Acesta are forma generală:

```
sizeof(tip)
```

Mai sus, *tip* reprezintă tipul căruia i se determină dimensiunea. Operatorul **sizeof** poate fi utilizat numai într-un context nesigur. Acesta este deci destinat unor situații speciale, întâlnite îndeosebi la amestecarea codului tratat cu cod netratat.

lock

Cuvântul cheie **lock** se utilizează atunci când se lucrează cu *fire multiple de execuție*. În C#, programele pot conține două sau mai multe *fire de execuție*. În acest caz, anumite operații ale programului sunt multiprogramate. Aceste operații din program se execută în mod independent și, din punct de vedere conceptual, simultan. Aceasta ridică perspectiva unui tip aparte de probleme: ce se întâmplă dacă două fire încearcă simultan să utilizeze o resursă care poate fi utilizată de un singur fir la un moment dat? Pentru a rezolva această problemă, trebuie să creați *secțiuni critice de cod*, care pot fi executate de un singur fir la un moment dat. Aceasta se realizează utilizând cuvântul cheie **lock**. Forma generală a acestuia este prezentată mai jos:

```
lock(obiect) {
// sectiune critica
}
```

Mai sus, *obiect* este un obiect care încearcă să execute secțiunea critică. Dacă un fir de execuție a intrat deja în secțiunea critică, atunci un al doilea fir va aștepta până când primul termină execuția secțiunii. Când i se acordă accesul, firul de execuție în așteptare poate executa secțiunea critică.

readonly

Este posibil să creați un câmp membru al unei clase accesibil numai la citire declarându-l ca **readonly**. Câmpurile **readonly** pot fi inițializate, după care nu mai pot fi modificate. Câmpurile **readonly** reprezintă deci o modalitate elegantă de a crea constante utilizate pe parcursul programului (ca de exemplu dimensiuni ale tablourilor etc.) Câmpurile **readonly** pot fi atât statice, cât și ne-statice.

Iată un exemplu care creează un câmp **readonly**:

```
// Prezentarea campurilor readonly.
using System;

class MyClass {
  public static readonly int SIZE = 10;
}

class DemoReadOnly {
  public static void Main() {
    int [] nums = new int[MyClass.SIZE];

    for(int i = 0; i < MyClass.SIZE; i++)
      nums[i] = i;

    foreach(int i in nums)
      Console.Write(i + " ");
    // MyClass.SIZE = 100;  // Eroare! – nu se poate modifica
  }
}
```

În esență, se declară o constantă.

Mai sus, câmpul **MyClass.SIZE** este inițializat cu 10. După aceasta, el poate fi utilizat, dar fără a fi modificat. Pentru a vă convinge de aceasta, încercați să eliminați comentariul de pe ultima linie, după care recompilați programul. După cum veți observa, se va produce o eroare.

stackalloc

Puteți aloca memorie direct pe stivă, utilizând **stackalloc**. Acesta se poate utiliza numai pentru inițializarea variabilelor locale și are următoarea formă generală:

```
tip *p = stackalloc tip[size]
```

Mai sus, *p* este un pointer care primește adresa unei zone de memorie suficiente pentru a memora un număr egal cu *size* de obiecte de tipul *tip*. **stackalloc** trebuie utilizat numai într-un context nesigur.

În mod obișnuit, memoria pentru obiecte este alocată în zona *heap*, care este o regiune care conține memoria disponibilă. Alocarea memoriei pe stivă reprezintă o excepție. Variabilele alocate pe stivă nu sunt supuse colectării automate. Ele există

doar pe parcursul execuției blocului în care sunt declarate. Singurul avantaj al utilizării lui **stackalloc** este că nu mai trebuie să vă preocupați de variabilele care pot fi mutate de mecanismul de colectare automată a spațiului.

Instrucțiunea using

În plus față de *directiva* **using** prezentată ceva mai devreme, **using** are și o a doua formă, care este denumită *instrucțiunea* **using**. Aceasta are forma generală:

```
using(obiect) {
// se utilizeaza obiectul
}
```

Mai sus, obiect este un obiect care este utilizat în interiorul blocului using. La sfârșitul execuției blocului, se va apela automat metoda **Dispose()**. Această formă a lui **using** funcționează numai pentru obiectele care implementează interfața **System.IDisposable.**

const și volatile

Modificatorul **const** se utilizează la declararea variabilelor care nu pot fi modificate. Aceste variabile trebuie să fie inițializate de la declarare. În concluzie, o variabilă **const** este de fapt o constantă. De exemplu,

```
const int i = 10;
```

creează o variabilă **const** numită **i**, care are valoarea 10.

Modificatorul **volatile** informează compilatorul că valoarea unei variabile se poate modifica într-o manieră care nu a fost precizată explicit în program. De exemplu, o variabilă care memorează timpul curent din sistem poate fi actualizată automat de către sistemul de operare. În această situație, conținutul variabilei este alterat fără nici o instrucțiune de atribuire explicită. Motivul pentru care modificarea externă a unei variabile are un rol important este acela că anumite expresii sunt optimizate automat de către compilatorul de C#, care presupune că o variabilă nu își schimbă valoarea dacă nu apare în partea stângă a unei instrucțiuni de atribuire. Dacă însă anumiți factori externi programului, cum ar fi un alt fir de execuție, modifică valoarea variabilei, presupunerea de mai sus este eronată. Utilizând **volatile**, informați compilatorul că trebuie să determine valoarea acestei variabile la fiecare acces.

Ce urmează?

Felicitări ! Dacă ați citit și ați încercat exemplele din cele 12 module de până acum, vă puteți numi deja programator în C#. Desigur, există încă multe lucruri de învățat despre C#, cum ar fi bibliotecile de funcții sau subsistemele limbajului, dar acum aveți un fundament solid pe care vă puteți construi cunoștințele și experiența. Iată numai câteva din subiectele pe care le puteți acum aprofunda:

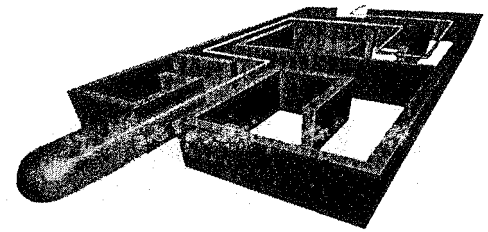
- Utilizarea limbajului C# la crearea aplicațiilor grafice Windows, care utilizează diferite elemente ale interfețelor cu utilizatorul, cum ar fi butoane, meniuri, liste sau bare de defilare.
- Crearea aplicațiilor cu mai multe fire de execuție.
- Construirea componentelor.
- Utilizarea claselor C# care reprezintă colecții, care implementează obiecte cum sunt stivele, cozile sau listele.
- Utilizarea facilităților limbajului C# pentru lucrul în rețea.
- Gestiunea versiunilor aplicațiilor dumneavoastră.

Pentru a vă continua studiul în C#, recomandăm lucrarea *„C# – Manual complet"* de același autor. Această lucrare prezintă toate subiectele menționate mai sus, precum și multe altele. În lucrare, veți găsi o prezentare completă a limbajului C#, alături de bibliotecile, subsistemele și aplicațiile sale.

Verificarea cunoștințelor

1. Arătați cum trebuie declarată o delegare numită **filter**, care întoarce o valoare **double** și primește un parametru **int**.
2. Cum se poate realiza facilitatea de multicast utilizând delegări? Ce restricții trebuie îndeplinite?
3. Ce legătură există între delegări și evenimente?
4. Este posibil ca evenimentele să fie multicast? Evenimentele sunt destinate instanțelor sau claselor în general?
5. Care este principalul avantaj al spațiilor de nume?
6. Prezentați forma cu alias a lui **using**.
7. Care sunt cele două tipuri de operatori de conversie? Prezentați formele generale ale acestora.
8. Denumiți directivele preprocesor care se utilizează pentru compilarea condiționată.
9. Prezentați sintaxa atributelor.
10. Ce este codul nesigur?
11. Cum puteți determina tipul unui obiect la momentul execuției?
12. Pe cont propriu, continuați studiul și experimentați facilitățile limbajului C#.

Anexa A
Răspunsurile testelor de verificare a cunoştinţelor



Modulul 1: Fundamentele limbajului C#

1. Ce este MSIL şi ce importanţă prezintă pentru C#?
   MSIL reprezintă prescurtarea de la Microsoft Intermediate Language. MSIL este un set optimizat şi portabil de instrucţiuni în limbaj de asamblare, care sunt compilate până la stadiul de cod executabil de compilatoarele JIT. Formatul MSIL asigură portabilitatea, securitatea şi compatibilitatea lucrului în limbaj mixt din C#.
2. Ce este motorul comun de programare?
   Motorul comun de programare (MCP) este partea arhitecturii .NET care gestionează execuţia programelor C# şi a celorlalte programe compatibile .NET.
3. Care sunt cele trei principii fundamentale ale programării orientate spre obiecte?
   Cele trei principii fundamentale ale programării orientate spre obiecte sunt încapsularea, polimorfismul şi moştenirea.
4. Cu ce începe execuţia programelor C#?
   Execuţia programelor C# începe cu metoda **`Main()`**.
5. Ce este o variabilă? Ce este un spaţiu de nume?
   O variabilă este o locaţie de memorie care are asociat un nume. Conţinutul unei variabile se poate modifica pe parcursul execuţiei unui program. Un spaţiu de nume este o regiune declarativă. Spaţiile de nume permit separarea unui set de nume de alte seturi de nume.
6. Care dintre următoarele nume de variabile sunt incorecte?
   A. count
   B. $count
   C. count27
   D. 67count
   E. @if
   Răspunsurile **B. $count** şi **D. 67count** sunt incorecte. Numele variabilelor nu pot începe cu semnul **$** sau cu cifre.
7. Cum puteţi crea un comentariu pe o singură linie? Cum puteţi crea un comentariu multilinie?

   Un comentariu pe o singură linie începe cu `//` şi se termină la sfârşitul liniei. Comentariile multilinie încep cu `/*` şi se termină cu `*/`.
8. Prezentaţi forma generală a instrucţiunii **`if`**. Prezentaţi forma generală a buclei **`for`**.
   Forma generală a lui **`if`**:
   `if`(*condiţie*) *instrucţiune*;
   Forma generală a lui **`for`**:
   `for`(*iniţializare*; *condiţie*; *iteraţie*) *instrucţiune*;
9. Cum puteţi crea un bloc de cod?
   Un bloc de cod începe cu o acoladă deschisă { şi se termină cu o acoladă închisă }.
10. Este obligatoriu ca fiecare program C# să înceapă cu următoarea instrucţiune?
    `using System;`
    Nu, instrucţiunea **`using System`** nu este obligatorie, dar omiterea ei înseamnă că trebuie să calificaţi toţi membrii spaţiului de nume **`System`**, precedându-i cu **`System`** (numele spaţiului).

Modulul 2: Prezentarea tipurilor de date şi operatorilor

1. De ce în C# domeniul şi comportamentul tipurilor simple sunt precizate strict?
   Limbajul C# precizează strict domeniul şi comportamentul tipurilor simple pentru asigurarea portabilităţii pe diverse platforme.
2. Care este tipul caracter în C#, şi care este diferenţa dintre acest tip şi tipul caracter utilizat în alte limbaje de programare?
   Tipul caracter din C# este **`char`**. Caracterele limbajului C# folosesc reprezentarea Unicode în locul setului ASCII, reprezentare adoptată şi de multe alte limbaje de programare.
3. O variabilă **`bool`** poate lua orice valoare, deoarece orice valoare nenulă reprezintă valoarea logică „adevărat". Este adevărat sau fals?
   Este fals. Valorile **`bool`** pot fi numai **`true`** sau **`false`**.
4. Se dă textul afişat de un program:
   ```
   One
   Two
   Three
   ```
   Utilizând un singur şir de caractere şi secvenţe escape, scrieţi instrucţiunea **`WriteLine()`** care a afişat acest text.
   ```
   Console.WriteLine("One\nTwo\nThree");
   ```
5. Unde este greşeala în secvenţa de mai jos?
   ```
   for(i = 0; i < 10; i++) {
     int sum;
   ```

```
  sum = sum + i;
}
Console.WriteLine("Suma este: " + sum);
```

Secvența de mai sus conține trei erori fundamentale. Mai întâi, variabila **sum** este creată de fiecare dată când se intră în blocul definit de **for** și distrusă la ieșirea din bloc. În consecință, variabila nu își păstrează valoarea de la o iterație la următoarea. Încercarea de a utiliza valoarea lui **sum** pentru a memora suma acumulată pe parcursul iterațiilor este deci inutilă. În al doilea rând, **sum** nu va fi cunoscută în afara blocului în care a fost declarată. Referirea variabilei în cadrul instrucțiunii **WriteLine()** este deci incorectă. În al treilea rând, variabila **sum** nu a fost inițializată.

6. Precizați diferența dintre forma prefixată și cea postfixată a operatorului de incrementare.
   Atunci când operatorul de incrementare își precede operandul, operația corespunzătoare este efectuată înainte ca valoarea operandului să fie obținută pentru a fi utilizată în restul expresiei. Dacă operatorul urmează operandului, valoarea operandului este obținută înainte de incrementare.
7. Arătați cum se poate utiliza un operator ȘI scurtcircuitat pentru a evita o eroare de împărțire la zero.

```
if((b != 0) && (val / b) ...
```

8. În cadrul unei expresii, la ce tip sunt promovate **byte** și **short**?
   Atunci când sunt utilizate în expresii, valorile **byte** și **short** sunt promovate la **int**.
9. Care dintre tipurile de mai jos nu poate fi combinat într-o expresie cu o valoare de tip **decimal**?
   A. float
   B. int
   C. uint
   D. byte
   Răspunsul este **A. float.**
10. În general, când este necesar un cast?
    Casturile sunt necesare la conversia între tipuri incompatibile sau atunci când conversia are loc prin restrângere.
11. Scrieți un program care determină toate numerele prime cuprinse între 1 și 100.
    Iată o modalitate de determinare a numerelor prime cuprinse între 1 și 100. Problema admite, desigur, și alte soluții.

```
// Determinarea numerelor prime cuprinse intre 1 si 100.

using System;

class Prime {
  static void Main() {
    int i, j;
    bool estePrim;

    for(i = 1; i < 100; i++) {
      estePrim = true;

      // verifica daca numarul se divide exact cu alt numar
      for(j = 2; j <= i/2; j++)
        // daca se divide, atunci nu este prim
        if((i % j) == 0) estePrim = false;

      if(estePrim)
        Console.WriteLine(i + " este prim.");
    }
  }
}
```


Modulul 3: Instrucțiuni de control

1. Scrieți un program care citește caractere de la tastatură până când se introduce un punct. Modificați programul astfel încât să numere spațiile introduse, afișând la sfârșit numărul lor.

```
// Numararea spatiilor.

using System;

class Spatii {
  public static void Main() {
    char ch;
    int spatii = 0;

    Console.WriteLine("Introduceti un punct pentru terminare.");

    do {
      ch = (char) Console.Read();
      if(ch == ' ') spatii++;
    } while(ch != '.');

    Console.WriteLine("Numarul de spatii: " + spatii);
  }
}
```

2. La instrucțiunea **switch**, secvența de cod de la un **case** se poate continua cu următorul?
   Nu. Regula de interzicere a execuției în secvență precizează că secvența de cod asociată unei etichete **case** nu poate continua cu următoarea. Instrucțiunile **case** pot fi însă „stivuite" (adică pot exista mai multe instrucțiuni **case** care să refere aceeași secvență de cod).
3. Prezentați forma generală a scării **if-else-if**.

if(*condiție*)
  instrucțiune;
else if(*condiție*)

instrucțiune;
  else if(*condiție*)
instrucțiune;

.

.

.

else
  instrucțiune;

4. Se dă secvența:

```
if(x < 10)
  if(y > 100) {
    if(! done) x = z;
    else y = z;
  }
else Console.WriteLine("eroare"); // care if?
```

Precizați cu care **if** se asociază ultimul **else**.
Ultimul **else** este asociat cu **if(y > 100)**, care este **if**-ul cel mai apropiat situat pe același nivel cu el.

5. Scrieți instrucțiunea **for** pentru o buclă care numără de la 1 000 la 0 cu pasul –2.

```
for(int i = 1000; i >= 0; i -= 2) // ...
```

6. Este corectă secvența următoare?

```
for(int i = 0; i < num; i++)
  sum += i;

count = i;
```

Nu, variabila **i** nu este cunoscută în afara instrucțiunii **for** în care a fost declarată.

7. Explicați care este rolul instrucțiunii **break**.
Instrucțiunea **break** determină terminarea buclei sau instrucțiunii **switch** celei mai apropiate care o conține.

8. În fragmentul următor, ce se afișează după execuția instrucțiunii **break**?

```
for(i = 0; i < 10; i++) {
  while(running) {
    if(x < y) break;
    //...
  }
  Console.WriteLine("dupa while");
}
Console.WriteLine("dupa for");
```

După execuția instrucțiunii **break**, se va afișa textul „dupa while".

9. Ce afișează următorul fragment de cod?

```
for(int i = 0; i < 10; i++) {
  Console.Write(i + " ");
  if((i%2) == 0) continue;
  Console.WriteLine();
}
```



```
0 1
2 3
4 5
6 7
8 9
```

10. Expresia de iterație dintr-o buclă **for** nu trebuie întotdeauna să modifice variabila de control a buclei cu o cantitate fixă. Variabila respectivă se poate modifica în mod arbitrar. Folosind acest concept, scrieți un program care utilizează o buclă **for** pentru generarea și afișarea progresiei geometrice 1, 2, 4, 8, 16, 32 etc.

```
/* Utilizarea unei bucle for pentru generarea progresiei:

   1, 2, 4, 8, 16, ...
*/

using System;

class Progresie {
  public static void Main() {

    for(int i = 1; i < 100; i += i)
      Console.Write(i + " ");
  }
}
```

11. Literele mici ale setului ASCII diferă de literele mari cu 32. Prin urmare, pentru a converti o literă mică în majusculă, se scade 32 din valoarea literei. Utilizați această informație pentru a scrie un program care citește caractere de la tastatură, schimbând literele mici în litere mari și invers, afișând totodată rezultatul modificării. Celelalte caractere nu vor fi modificate. Programul va citi caractere până când utilizatorul tastează un punct. La final, programul va afișa numărul modificărilor efectuate.

```
// Schimbarea literelor mici in litere mari si invers.

using System;

class CaseChg {
  public static void Main() {
    char ch;
    int changes = 0;

    Console.WriteLine("Introduceti un punct pentru terminare.");

    do {
      ch = (char) Console.Read();
      if(ch >= 'a' & ch <= 'z') {
        ch -= (char) 32;
        changes++;
        Console.WriteLne(ch);
      }
      else if(ch >= 'A' & ch <= 'Z') {
```

```
            ch += (char) 32;
            changes++;
            Console.WriteLine(ch);
          }
        } while(ch != '.');
        Console.WriteLine("Numarul de modificari: " + changes);
      }
    }
```

Modulul 4: Prezentarea claselor, obiectelor și metodelor

1. Ce diferență există între o clasă și un obiect?
   Clasele sunt concepte abstracte, care descriu forma și comportamentul obiectelor. Obiectele reprezintă instanțe fizice ale claselor.
2. Cum poate fi definită o clasă?
   Clasele sunt definite utilizând cuvântul cheie **class**. În interiorul instrucțiunii **class**, sunt specificate datele și codul care alcătuiesc clasa.
3. Ce entități au câte o copie în cadrul fiecărui obiect?
   Fiecare obiect deține o copie proprie a variabilelor instanță definite în cadrul clasei sale.
4. Utilizând două instrucțiuni separate, arătați cum puteți declara un obiect numit **counter** aparținând unei clase numite **MyCounter**.
   ```
   MyCounter counter;
   counter = new MyCounter();
   ```
5. Arătați cum puteți declara o metodă numită **myMeth()** care întoarce un rezultat de tipul **double** și are doi parametri de tipul **int**, numiți **a** și **b**.
   ```
   double myMeth(int a, int b) { // ...
   ```
6. Cum se efectuează revenirea dintr-o metodă care întoarce o valoare?
   O metodă care întoarce o valoare trebuie să revină executând o instrucțiune **return**, care transmite valoarea întoarsă procesului apelant.
7. Ce nume trebuie să aibă un constructor?
   Constructorii au același nume cu clasa în care sunt definiți.
8. Ce face operatorul **new**?
   Operatorul **new** alocă memorie pentru un obiect și inițializează obiectul utilizând constructorul acestuia.
9. Ce este colectarea spațiului neutilizat și cum funcționează? Ce este un destructor?
   Colectarea spațiului neutilizat este mecanismul prin care obiectele nefolosite sunt reciclate, astfel încât memoria ocupată de ele să poată fi reutilizată. Destructorul este o metodă apelată exact înainte ca un obiect să fie reciclat.

10. Ce reprezintă **this**?
    Cuvântul cheie **this** reprezintă o referință către obiectul pentru care are loc invocarea unei metode. Acest obiect este implicit transmis metodei.

Modulul 5: Mai multe despre tipurile de date și operatori

1. Arătați cum se declară un tablou unidimensional cu 12 elemente de tipul **double**.
   ```
   double [] x = new double[12];
   ```
2. Declarați un tablou bidimensional cu elemente de tipul **int** și cu dimensiunile 4 x 5.
   ```
   int [,] nums = new int[4, 5];
   ```
3. Declarați un tablou bidimensional în scară de tipul **int**, în care prima dimensiune este 5.
   ```
   int [] [] nums = new int [5] [];
   ```
4. Inițializați un tablou unidimensional de tipul **int** cu numerele de la 1 la 5.
   ```
   int [] x = {1, 2, 3, 4, 5 };
   ```
5. Prezentați bucla **foreach**. Arătați forma generală a instrucțiunii.
   Bucla **foreach** parcurge elementele unei colecții, întorcând pe rând câte un element. Forma generală a buclei este prezentată mai jos:

   foreach(*tip nume-var* **in** *colecție*) *instrucțiune*;
6. Scrieți un program care utilizează un tablou pentru a determina media a 10 valori de tipul **double**. Utilizați orice set de valori doriți.
   ```
   // Media a 10 valori double.
   using System;

   class Avg {
     public static void Main() {
       double [] nums = {1.1, 2.2, 3.3, 4.4, 5.5,
                         6.6, 7.7, 8.8, 9.9, 10.1 };

       double sum = 0;

       for(int i = 0; i < nums.Length; i++)
         sum += nums[i];

       Console.WriteLine("Media: " + sum / nums.Length);
     }
   }
   ```
7. Modificați programul de sortare din proiectul 5-1 astfel încât să sorteze un tablou de stringuri. Luați un exemplu pentru a demonstra că programul funcționează.

```
// Sortarea stringurilor prin metoda bulelor.

using System;

class StrBubble {
  public static void Main() {
    string strs = {
                    "acesta", "este", "un", "test",
                    "de", "sortare", "a", "stringurilor"
                  };
    int a, b;
    string t;
    int size;

    size = strs.Length; // numarul elementelor de sortat

    // afisarea tabloului initial
    Console.Write("Tabloul initial: ");
    for(int i = 0; i < size; i++)
      Console.Write(" " + strs[i]);
    Console.WriteLine();

    // sortarea stringurilor prin metoda bulelor
    for(a = 1; a < size; a++)
      for(b = size - 1; b >= a; b--) {
        if(strs[b-1].CompareTo(strs[b]) > 0) { // ordinea este incorecta
          t = strs[b-1];
          strs[b-1] = strs[b];
          strs[b] = t;
        }
      }

    // afisarea tabloului sortat
    Console.Write("Tabloul sortat: ");
    for(int i = 0; i < size; i++)
      Console.Write(" " + strs[i]);
    Console.WriteLine();
  }
}
```

8. Ce diferență este între metodele **IndexOf()** și **LastIndexOf()** din clasa **string**?
   Metoda **IndexOf()** găsește prima apariție a subșirului specificat, în timp ce **LastIndexOf()** determină ultima apariție.
9. Modificați clasa **Encode** care realizează criptarea unui mesaj astfel încât să utilizeze un **string** de 8 caractere pe post de cheie de criptare.

```
// Un cifru XOR imbunatatit.

using System;

class Encode {
  public static void Main() {
    string msg = "Acesta este un test";
    string encmsg = "";
    string decmsg = "";
    string key = "abcdefgi";
    int j;
```



```
    Console.Write("Mesajul initial: ");
    Console.WriteLine(msg);

    // codificarea mesajului
    j = 0;
    for(int i = 0; i < msg.Length; i++) {
      encmsg = encmsg + (char) (msg[i] ^ key[j]);
      j++;
      if(j == 8) j = 0;
    }

    Console.Write("Mesajul codificat: ");
    Console.WriteLine(encmsg);

    // decodificarea mesajului
    j = 0;
    for(int i = 0; i < msg.Length; i++) {
      decmsg = decmsg + (char) (encmsg[i] ^ key[j]);
      j++;
      if(j == 8) j = 0;
    }

    Console.Write("Mesajul decodificat: ");
    Console.WriteLine(decmsg);

  }
}
```

10. Se pot utiliza operatorii la nivel de biți pe valori de tipul **double**?
    Nu.
11. Arătați cum se poate rescrie secvența de mai jos utilizând operatorul ?.

```
if(x < 0) y = 10;
else y = 20;
```

Răspunsul este:

```
y = x < 0 ? 10 : 20;
```

12. În fragmentul următor, **&** reprezintă un operator la nivel de biți sau un operator logic? De ce?

```
bool a, b;
// ...
if(a & b) ...
```

Aici, **&** reprezintă un operator logic, deoarece operanzii sunt de tipul **bool**.

Modulul 6: O privire mai detaliată asupra metodelor și claselor

1. Dându-se fragmentul:

```
class X {
  int count;
```

este corectă secvența următoare?

```
class Y {
  public static void Main() {
    X ob = new X();

    ob.count = 10;
```

Nu, deoarece membrii privați nu sunt accesibili în afara clasei în care sunt definiți. După cum am precizat, membrii pentru care nu apare nici un specificator de acces sunt implicit privați.

2. Un specificator de acces trebuie să ______________ declarația unui membru.
<u>precedă</u>

3. O structură complementară cozii este structura de stivă. Aceasta folosește disciplina de acces „primul sosit, ultimul servit", fiind de multe ori similară cu un teanc de farfurii. Prima farfurie pusă pe masă este ultima care va fi utilizată. Creați o clasă numită **Stack** care implementează o stivă care memorează caractere. Denumiți metodele de acces la stivă **push()** și **pop()**. Permiteți utilizatorului să specifice dimensiunea stivei la creare. Ceilalți membri ai clasei **Stack** trebuie să rămână privați. Indicație: Puteți utiliza clasa **Queue** ca model; trebuie doar să schimbați maniera de acces la date.

```
// O clasa care reprezinta o stiva de caractere.

using System;

class Stack {
  char [] stck; // tabloul care memoreaza stiva
  int tos;      // varful stivei

  // Construieste o stiva vida cu dimensiunea precizata.
  public Stack(int size) {
    stck = new char[size];  // aloca memorie pentru stiva
    tos = 0;
  }

  // Construieste un obiect Stack "clonand" un altul.
  public Stack(Stack ob) {
    tos = ob.tos;
    stck = new char[ob.stck.Length];

    // copierea elementelor
    for(int i = 0; i < tos; i++)
      stck[i] = ob.stck[i];
  }

  // Construieste o stiva cu valori initiale date.
  public Stack(char[] a) {
    stck = new char[a.Length];

    for(int i = 0; i < a.Length; i++) {
      push(a[i]);
    }
  }
```



```
  // Introduce un caracter in stiva.
  public void push(char ch) {
    if(tos == stck.Length) {
      Console.WriteLine(" - Stiva este plina.");
      return;
    }

    stack[tos] = ch;
    tos++;
  }

  // Extrage un caracter din stiva.
  public char pop() {
    if(tos == 0) {
      Console.WriteLine(" - Stiva este vida.");
      return (char) 0;
    }

    tos--;
    return stck[tos];
  }
}

// Utilizeaza clasa Stack.
class SDemo {
  public static void Main() {
    // construim o stiva vida cu 10 elemente
    Stack stk1 = new Stack(10);

    char [] name = {"I", "o", "n"};

    // construim o stiva utilizand tabloul
    Stack stk2 = new Stack(name);
    char ch;
    int i;

    // inseram cateva caractere in stk1
    for(i = 0; i < 10; i++)
      stk1.push((char) ('A' + i));

    // construim o stiva utilizand una existenta
    Stack stk3 = new Stack(stk1);

    // afisam stivele.
    Console.Write("Stiva stk1: ");
    for(i = 0; i < 10; i++) {
      ch = stk1.pop();
      Console.Write(ch);
    }

    Console.WriteLine("\n");

    Console.Write("Stiva stk2: ");
    for(i = 0; i < 3; i++) {
      ch = stk2.pop();
      Console.Write(ch);
    }

    Console.WriteLine("\n");
```

```
    Console.Write("Stiva stk3: ");
    for(i = 0; i < 10; i++) {
      ch = stk3.pop();
      Console.Write(ch);
    }
  }
}
```

Iată rezultatul pe care programul îl va afişa:

```
Stiva stk1: JIHGFEDCBA

Stiva stk2: noI

Stiva stk3: JIHGFEDCBA
```

4. Se dă clasa:

```
class Test {
  int a;
  Test(int i) { a = i; }
}
```

Să se scrie o metodă numită **swap()** care interschimbă conţinutul a două obiecte de tipul **Test.**

```
void swap(Test ob1, Test ob2) {
  int t;

  t = ob1.a;
  ob1.a = ob2.a;
  ob2.a = t;
}
```

5. Este următorul fragment corect?

```
class X {
  int meth(int a, int b) { ... }
  string meth(int a, int b) { ... }
```

Nu. Metodele supraîncărcate pot întoarce rezultate de tipuri diferite, dar aceasta nu are nici o importanţă în procesul de rezoluţie. Metodele supraîncărcate *trebuie* să difere prin listele de parametri.

6. Scrieţi o metodă recursivă care afişează conţinutul unui **string** în ordine inversă.

```
// Afisarea unui string in ordine inversa utilizand recursivitatea.

using System;

class Invers {
  string str;

  public Invers(string s) {
    str = s;
  }
  public void invers(int idx) {
    if(idx != str.Length - 1) invers(idx + 1);

    Console.Write(str[idx]);
  }
}

class InvDemo {
  public static void Main() {
    Invers s = new Invers("Acesta este un test.");

    s.invers(0);
  }
}
```


7. Dacă o variabilă trebuie să fie partajată de toate instanţele unei clase, cum trebuie declarată?
   Variabilele partajate trebuie declarate ca **static.**
8. Ce rol au **ref** şi **out**? Ce diferenţă există între ei?
   Modificatorul **ref** forţează transferul unui parametru prin referinţă. Aceasta permite unei metode să modifice conţinutul parametrilor de apel. Parametrii **ref** primesc valori calculate în interiorul metodei. Modificatorul **out** este similar cu **ref**, dar nu poate fi utilizat pentru a transmite o valoare unei metode.
9. Prezentaţi cele patru forme ale metodei **Main()**.

```
public static void Main()
public static void Main(string[] args)
public static int Main()
public static int Main(string[] args)
```

10. Dându-se fragmentul de mai jos, care din următoarele apeluri sunt corecte?

```
void meth(int i, int j, params int[] args) { // ...
```

```
A.  meth(10, 12, 19);
B.  meth(10, 12, 19, 100);
C.  meth(10, 12, 19, 100, 200);
D.  meth(10, 12);
```

Toate apelurile sunt corecte.

Modulul 7: Supraîncărcarea operatorilor, indexări şi proprietăţi

1. Prezentaţi forma generală utilizată la supraîncărcarea unui operator unar. Ce tip trebuie să aibă parametrul metodei operator?

public static *tip-rez* **operator** *op*(*tip-param operand*)
{
  // operaţii
}

Operandul trebuie să aibă acelaşi tip cu clasa pentru care are loc supraîncărcarea operatorului.